McLab Tutorial
www.sable.mcgill.ca/mclab

Laurie Hendren, Rahul Garg and Nurudeen Lameed

Other McLab team members:
Andrew Casey, Jesse Doherty, Anton Dubrau, Jun Li, Amina Aslam, Toheed Aslam, Maxime Chevalier-Boisvert, Soroush Radpour, Oliver Savary , Maja Frydrychowicz, Clark Verbrugge

Sable Research Group
School of Computer Science
McGill University, Montreal, Canada

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 1 Intro - 1

This tutorial is intended to provide an overview of the challenges of compiling MATLAB and the tools provided by McGill's McLab project. Please feel free to reuse these slides, however please make sure you credit the authors of the slides and that you indicate the source of the original slides.

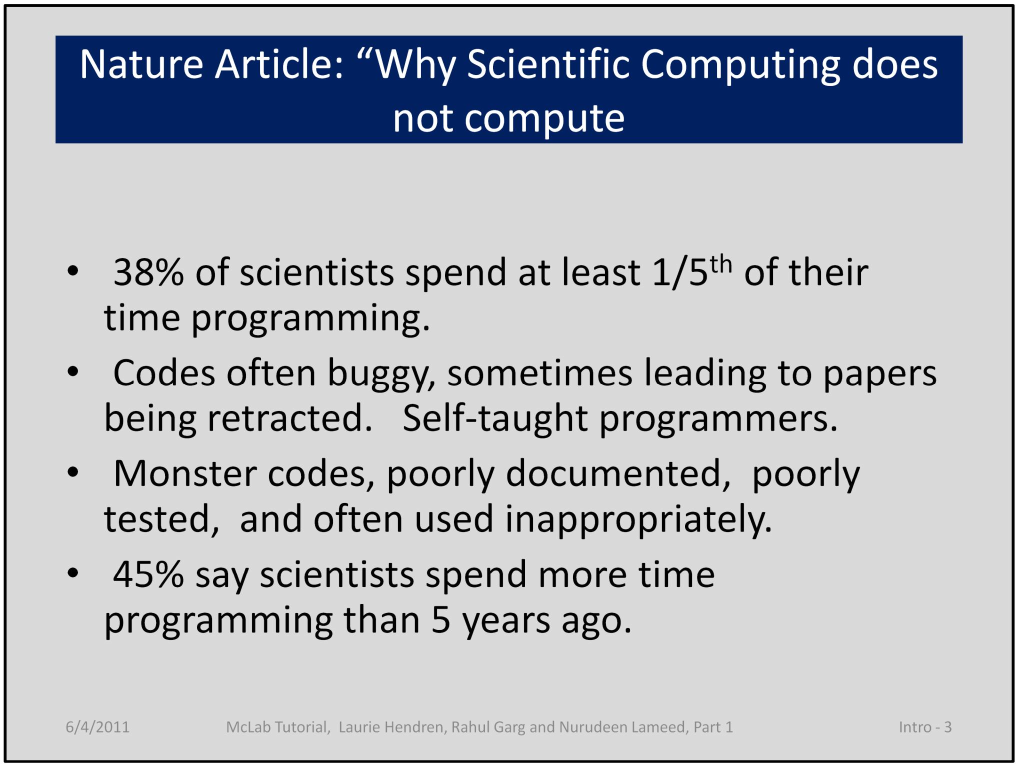
Tutorial Overview

- Why MATLAB?
- Introduction to MATLAB – challenges
- Overview of the McLab tools
  - Introduction to the front-end and extensions
  - IRs, Flow analysis framework and examples
  - Back-ends including the McVM virtual machine
- Wrap-up

This tutorial starts with an exploration of why it is important for compiler/PL researchers to work on MATLAB and languages like MATLAB.

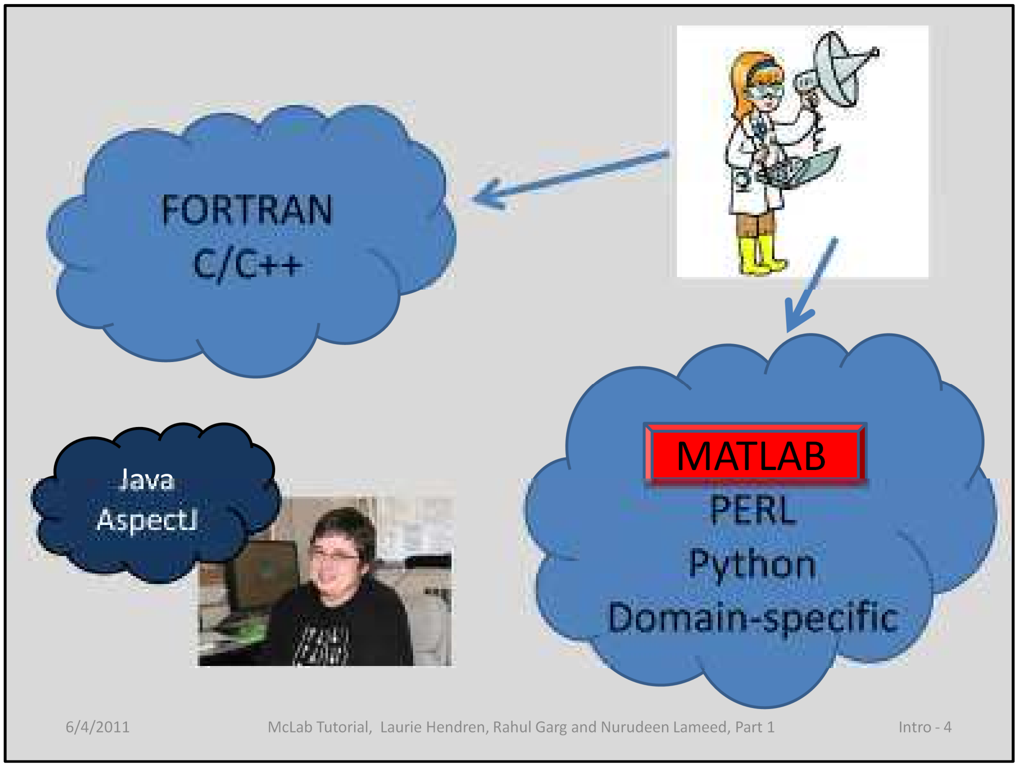
We then proceed to an introduction to the MATLAB language, and we illustrate some of the challenges of dealing with MATLAB.

The main body of the tutorial is composed of an introduction to the McLab toolset. We will give an introduction to the front-end and how it can be used to build MATLAB extensions, then we introduce our two IRs, McAST a high-level AST and McLAST a lower-level AST. We then move to an overview of our back-ends, with a particular focus on McVM and McJIT. Finally, we will give a short wrap-up.



October 2010 article in Nature, by Zeeya Merali. Survey of 2000 scientists.

It is important that compiler/PL researchers aim to provide programming languages and systems that both provide:
- programming environments in which scientists can program easily
- systems that lead to solid and extensible code.



Scientist in upper right ... Many different applications to program, which language to pick? Increasingly picking dynamic or scripting languages. Many scientific and engineering computations use MATLAB.

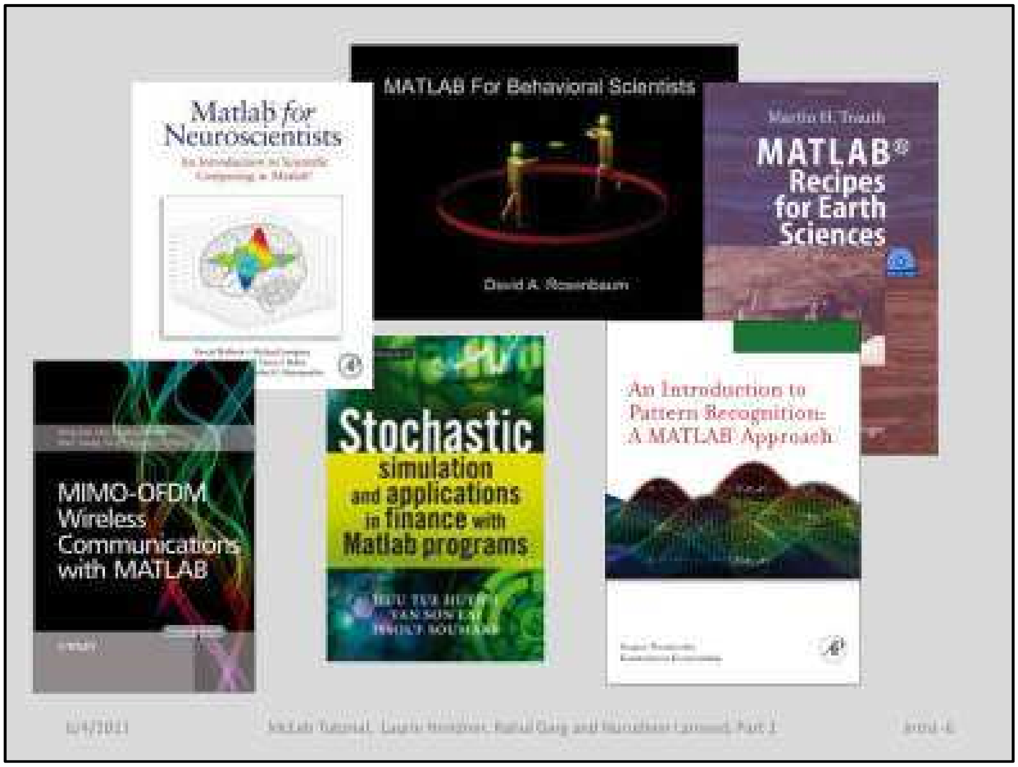
Computer Scientist, Compiler writer, lower left. Has worked on compilers and tools for object-oriented and aspect-oriented languages ... But scientists are not interested in these languages.

A lot of MATLAB programmers!

- Started as an interface to standard FORTRAN libraries for use by students.... but now
  - 1 million MATLAB programmers in 2004, number doubling every 1.5 to 2 years.
  - over 1200 MATLAB/Simulink books
  - used in many sciences and engineering disciplines
- Even more “unofficial” MATLAB programmers including those using free systems such as Octave or SciLab.

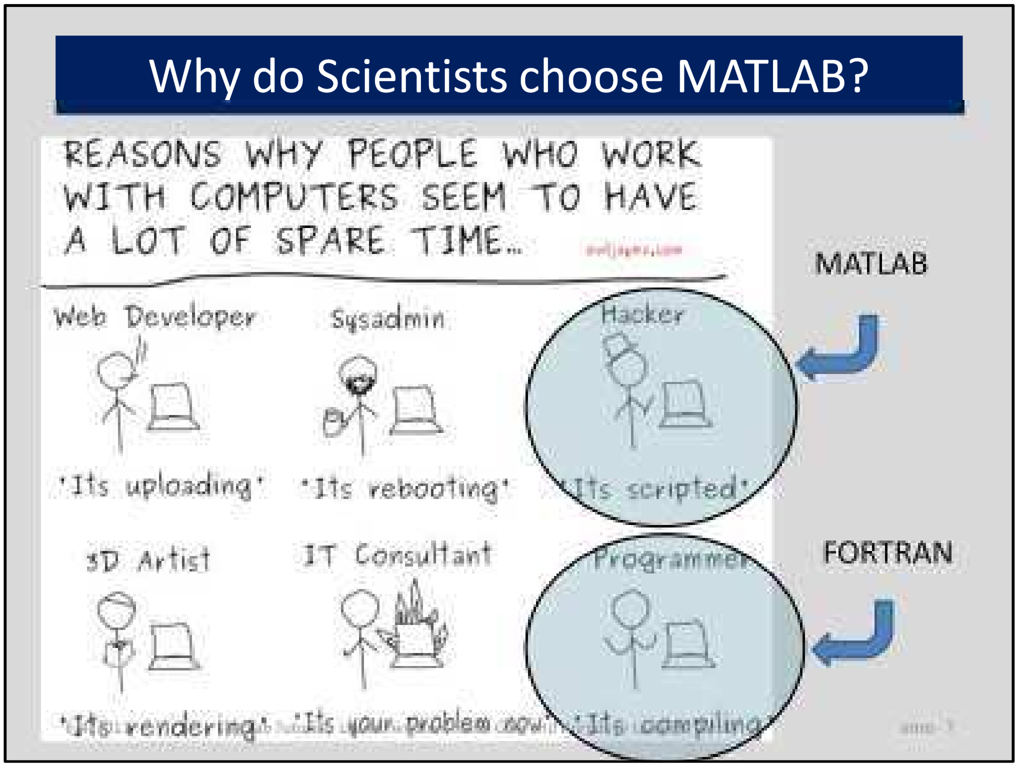
6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 1 Intro - 5

There are a lot of MATLAB users, shouldn't we be doing something for them?



Check out the number and variety of disciplines ….

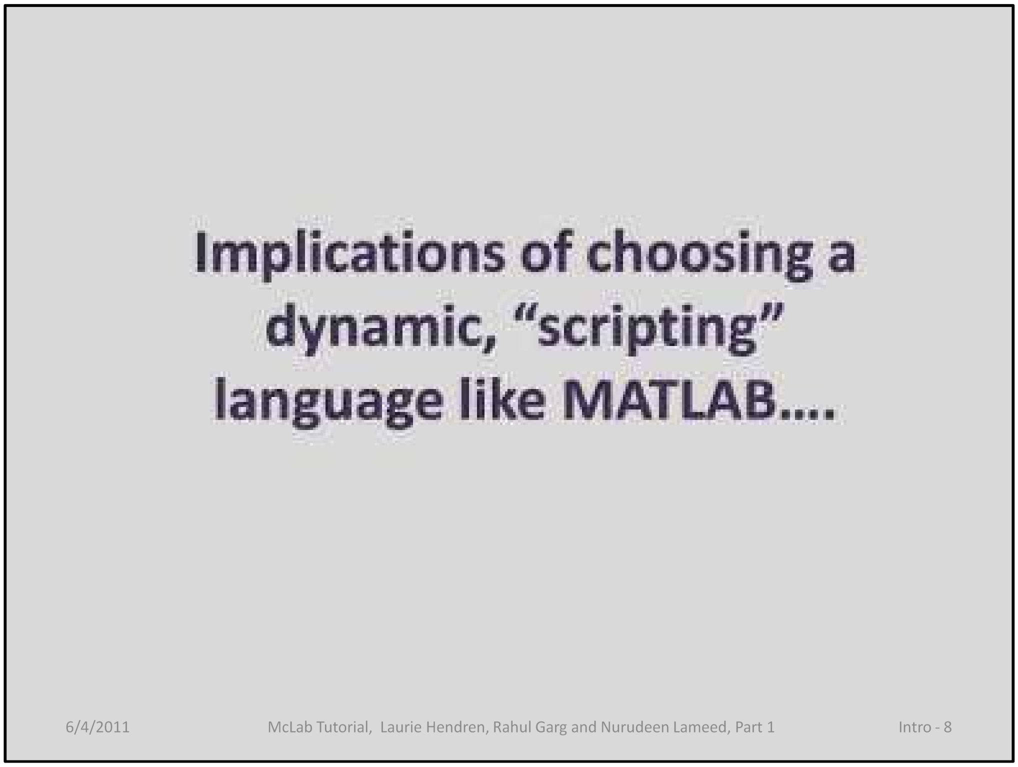
Books are often "how to" in terms of using MATLAB. We also need some books that describe MATLAB in way that both uses solid PL terminology and foundations, but also talks about the domain-specific applications.



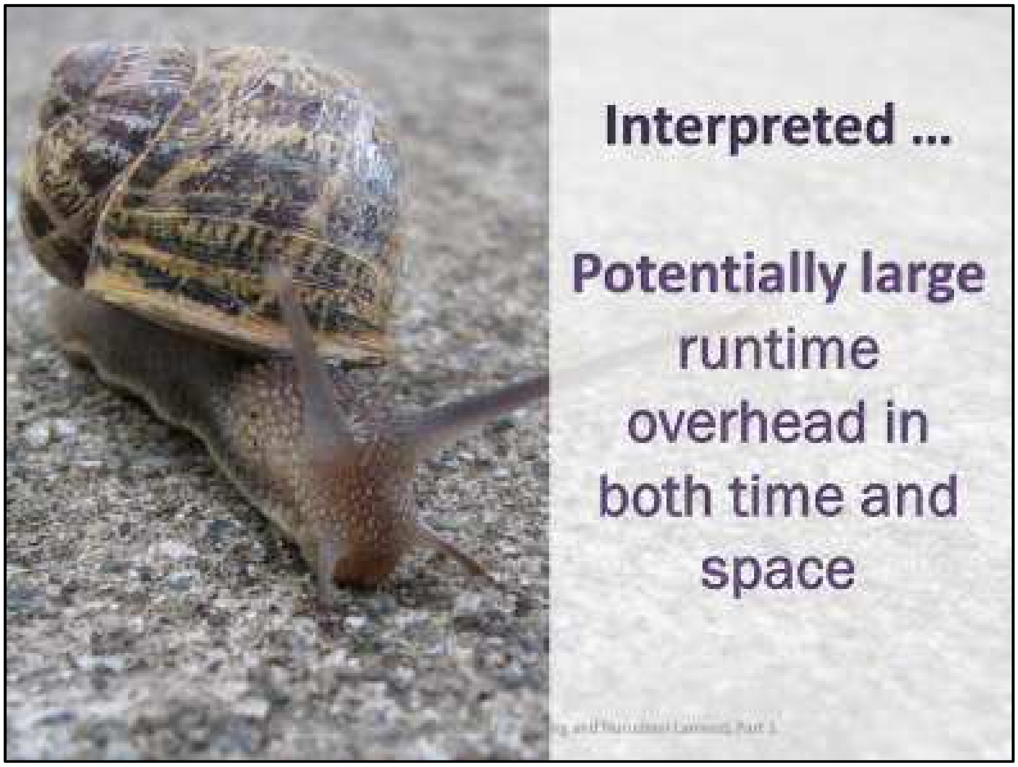
Why do scientists choose MATLAB?

Why not something like FORTRAN? - advantages are good compilers, efficient execution.

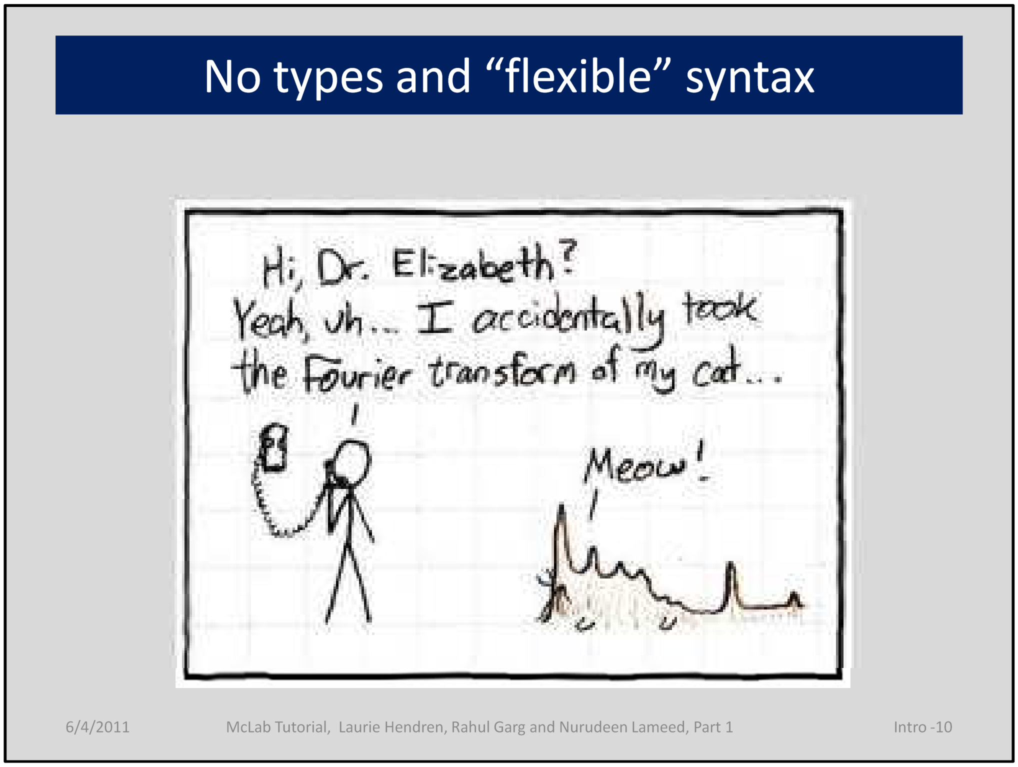
But programmers are choosing MATAB – faster prototyping – no types, lots of toolboxes, interactive development style...



Although Scientists like the interactive and "wild west" development style of MATLAB, what are the implications of choosing a dynamic "scripting" language like MATLAB?

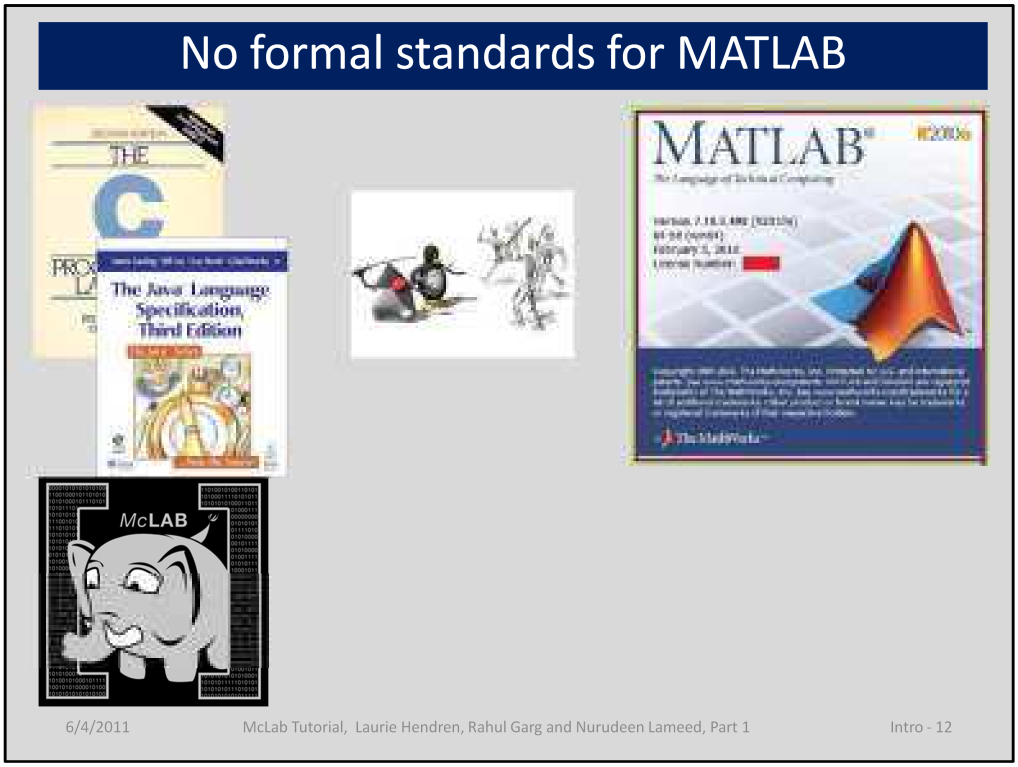


Original implementation by MATHWORKs interpreted, their system now contains a JIT (which they call an "accelerater"). Open implementations like Octave and Scilab are interepreted.



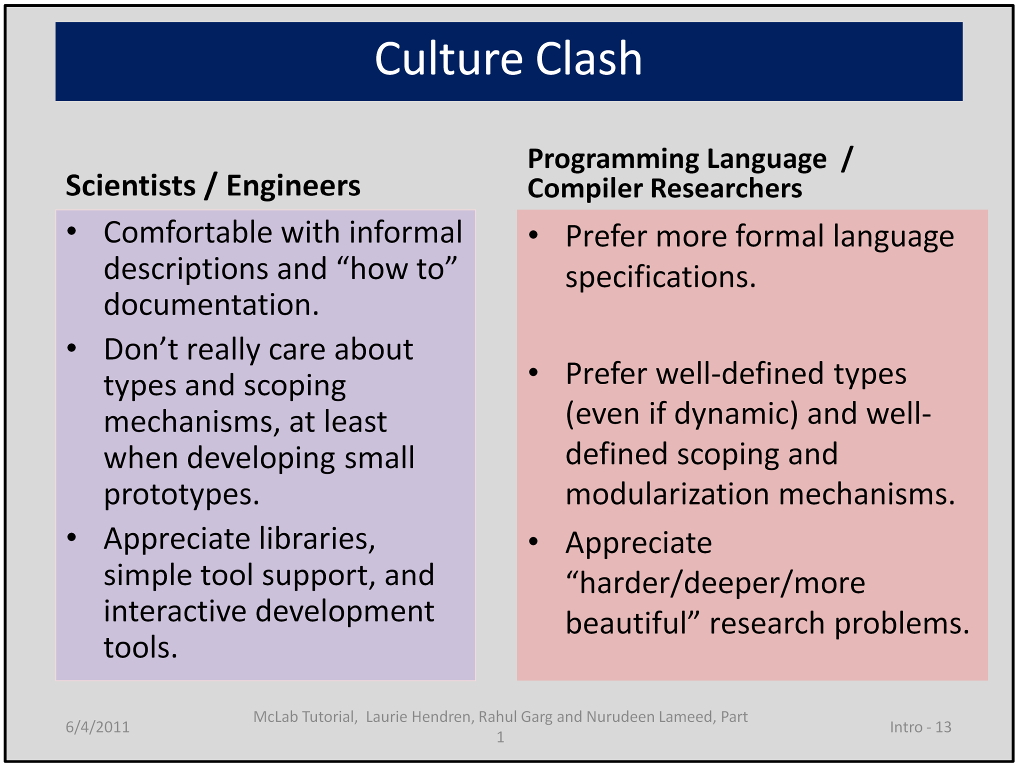
MATLAB often computes something, even if it was not was intended.

MATLAB programmers get very few static guarantees, but quite often program in some dynamic checks.



Lack of a standard – the semantics can change in a new release from Mathworks.

If the research community can help distill out a proper specification, it will enhance research opportunities and perhaps encourage some standardization.



PL and compiler researchers need to consider the scientific community and their perspective.

What can we do to enhance their programming experience, while still doing interesting research from a CS perspective?

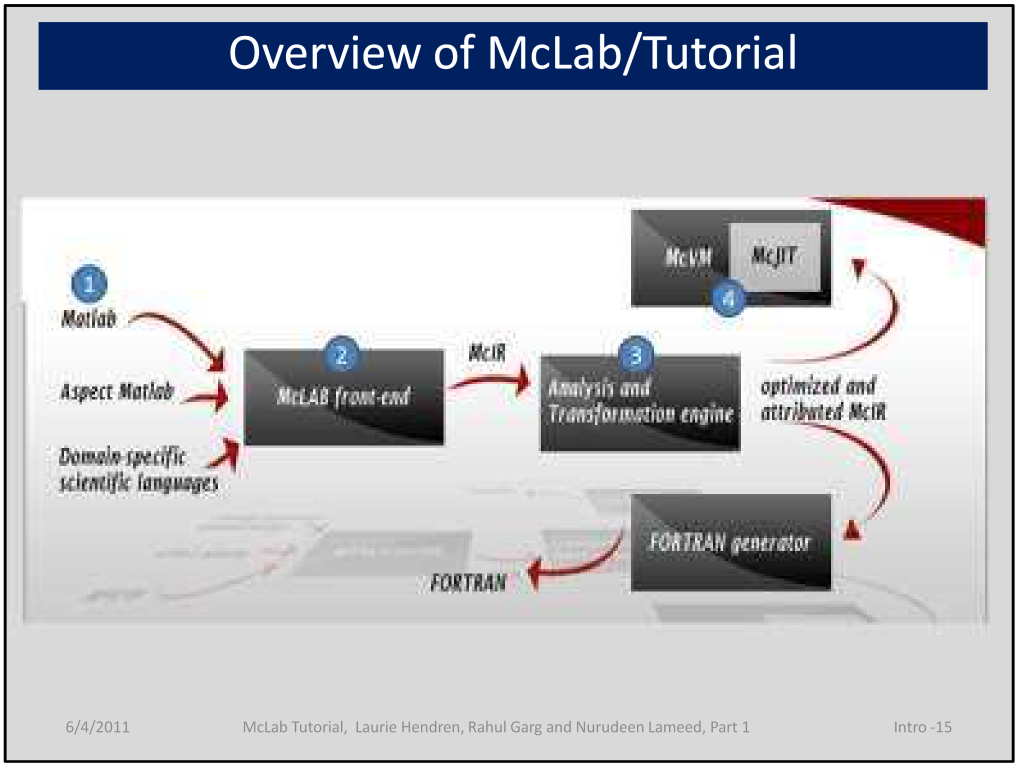
Does the PL/compiler community need to broaden their perspective of what is useful/good research?

Goals of the McLab Project

- Improve the understanding and documentation of the semantics of MATLAB.
- Provide front-end compiler tools suitable for MATLAB and language extensions of MATLAB.
- Provide a flow-analysis framework and a suite of analyses suitable for a wide range of compiler/soft. eng. applications.
- Provide back-ends that enable experimentation with JIT and ahead-of-time compilation.

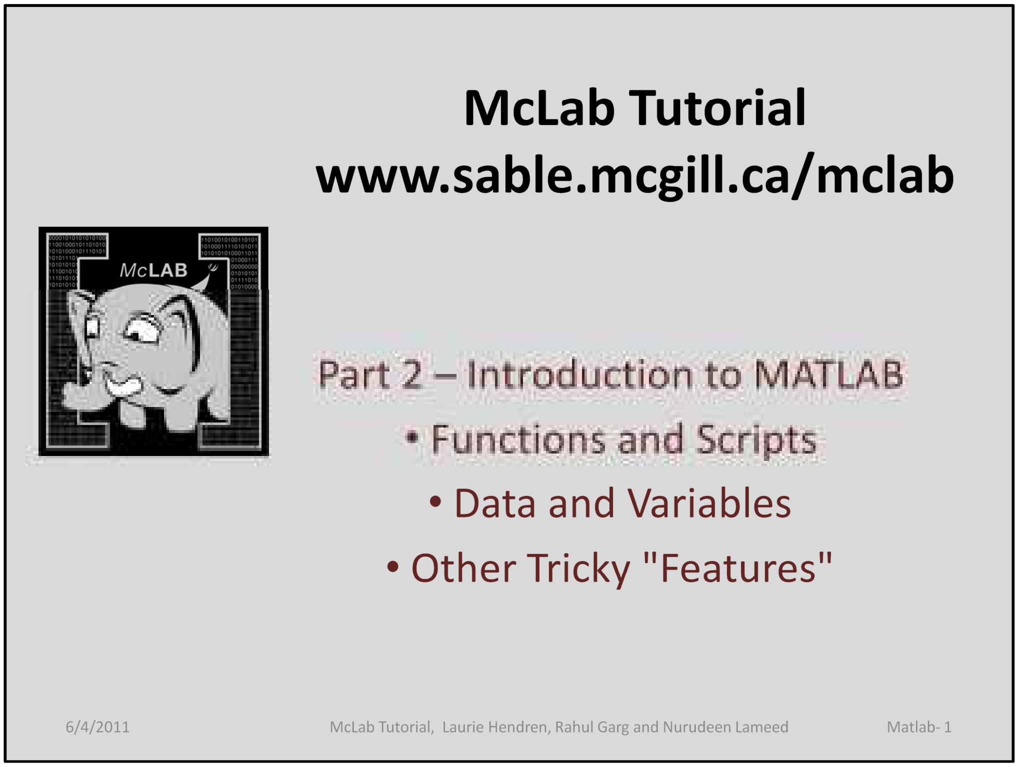
6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 1 Intro - 14

Our goals are to provide an infrastructure that supports the research for MATLAB, and to do such research ourselves.

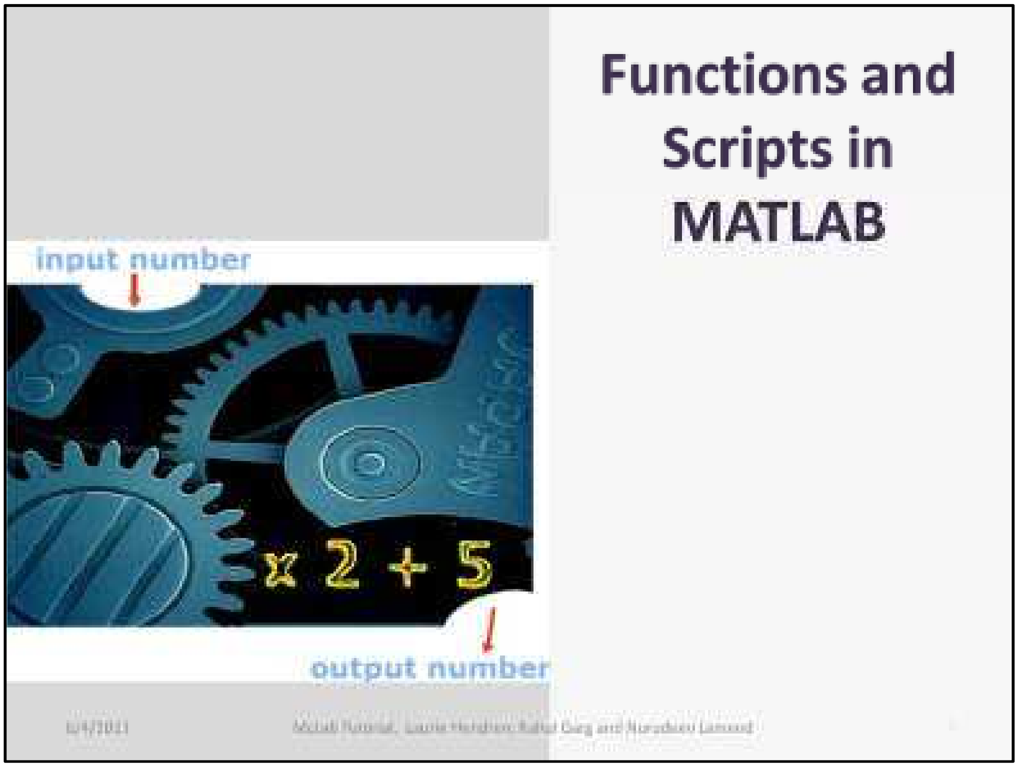


The rest of this tutorial will:

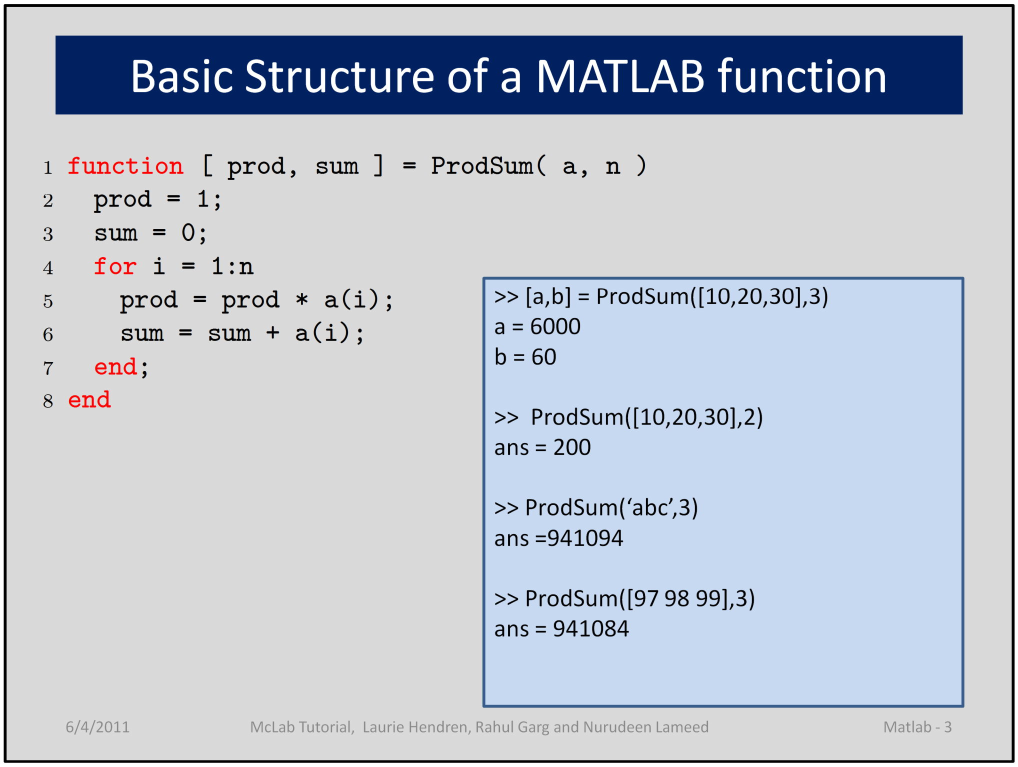
(1) Introduce MATLAB
(2) Give an overview of the McLab front-end and a small example of an extension
(3) Give an overview of the IRs and the Analysis Framework
(4) Discuss the back-ends, concentrating on McVM/McJIT



Before we can understand the tools, we first need to understand the MATLAB language. We have distilled out the important parts, and will first introduce functions and scripts, and then introduce data and variables. We then discuss some Matlab-specific tricky features.



There are two main ways of defining MATLAB computations, through functions which have input and output parameters, and through scripts which have no parameters, and are effectively just a sequence of statements. Let's look at functions first.



On first click:
Here is an example of a function definition of a function called ProdSum. It has two input parameters, "a" and "n", and two output parameters, "prod" and "sum". There is also a local variable "i".

This function should be stored in a file called ProdSum.m. If it is stored in a file of another name, say "foo.m" then the function can only be called as foo(...).

Note that there are no type declarations and no explicit declarations of variables. "i" is a variable because it is defined (i.e. it is assigned to in the header of the for loop.

There are no return statements for returning the values of the output parameters, the values returned are simply the values those variables contain when the function returns.

On second click:
Here is an example of an interaction with the MATLAB read-eval-print loop. The user types in the statement after the ">>" prompt and the statement is evaluated and the result printed. If an explicit assignment is made in the statement, the values of the lhs are printed, otherwise the result of evaluating the expression is assigned to a variable called "ans" and its value is printed.

The first call calls ProdSum with a 1x3 array [10,20,30] as the first argument, and a 1x1 array 3 as the second argument. The first return value is assigned to "a" and the second to "b".

The second call does not give a lhs, so the first of the return values is assigned to ans and the 2nd return value is thrown away. A similar thing would happen if you explicity said "myans = ProdSum....".

The third call illustrates that ProdSum works on various types. In this case the first

Basic Structure of a MATLAB function (2)

```
function [ prod, sum ] = ProdSum( a, n )
  prod = 1;
  sum = 0;
  for i = 1:n
    prod = prod * a(i);
    sum = sum + a(i);
  end;
end
```

```
>> [a,b] = ProdSum(@sin,3)
a = 0.1080
b = 1.8919

>>  [a,b] = ProdSum(@(x)(x),3)
a = 6
b = 6

>> magic(3)
ans = 8  1  6
      3  5  7
      4  9  2

>>ProdSum(ans,3)
ans=96
```

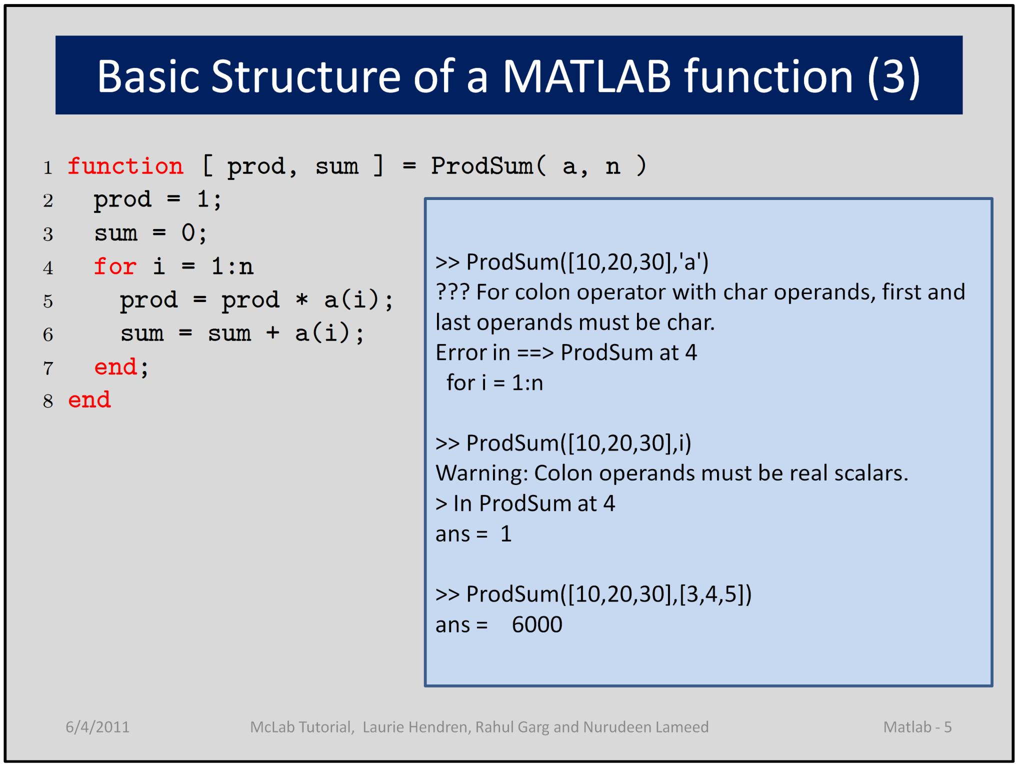
6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 4

1st click: On the previous slide we saw that the first argument could be 1xn arrays of different types. What else is allowed?

2nd click: The first statement shows a call using a function handle, @sin. This changes the meaning of a(i) in the body, as now it means an indirect call to sin(i). Note that the syntax of an indirect call is the same as the syntax for a direct call and an array index.

The 2nd statement illustrates an anonymous function, in this case just the identity function.

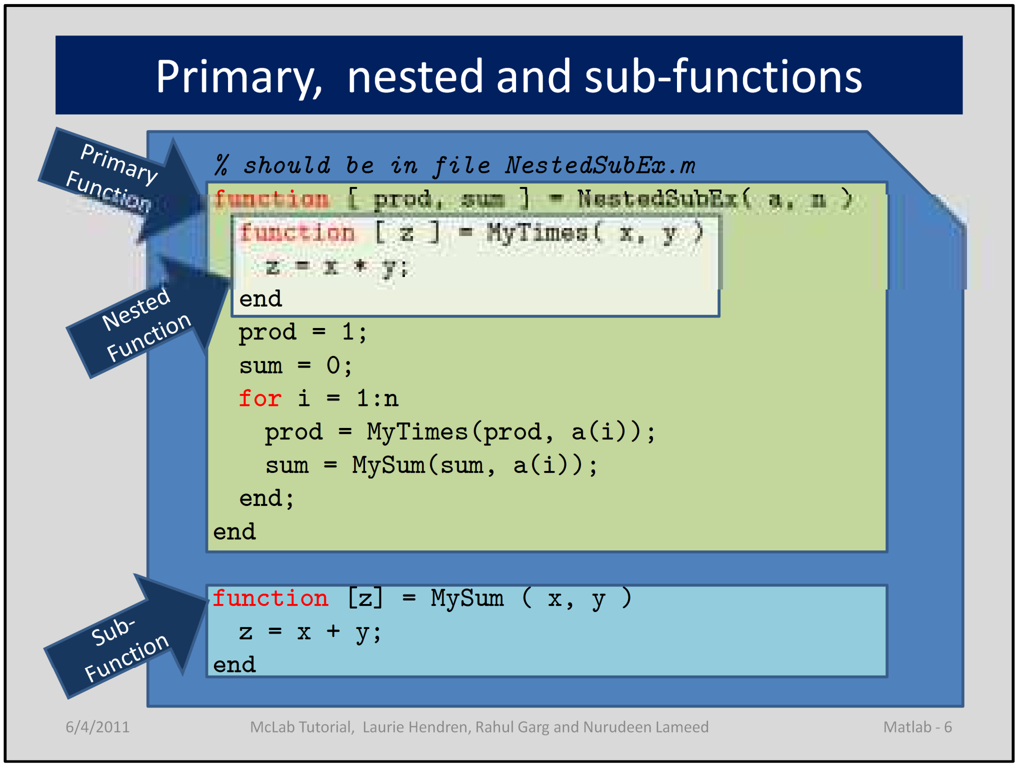
The 3rd and 4th statements illustrate what happens when you provide an array with higher dimensions that what is required by an indexing statement. In this case we create a 3x3 magic square, using the built-in function "magic" and then provide that as the 1st argument to "ProdSum". The indexing expression a(i) will still work, but it will automatically use 1 for all the missing dimensions. So, in this case it is effectively a(i,1), and the result is the product of the first column of a.



1st click: although MATLAB tolerates many different inputs, not all inputs will result in an answer, some will trigger errors, others will trigger warnings.
2nd click: For example, what happens when we give a character value for n, we might expect this to work, since characters were valid for the first argument. However, this causes a run-time error, since the colon operator only works with characters if both the left and right operands are characters. This is unlike the * and + operators, which tolerate arguments of different types.

The 2nd statement using "i" as the 2nd argument. What does this mean. In this case, it means the library function i, which returns the complex value 0+1i. In this case a warning is issued at run-time saying the colon wants real scalars, but it happily continues on assuming the real value 0.

Based on the previous error message we might expect that the colon operator only wants scalars, however if we try giving it [3,4,5], then it happily just uses the first element, 3, and produces the product of [10,20,30].

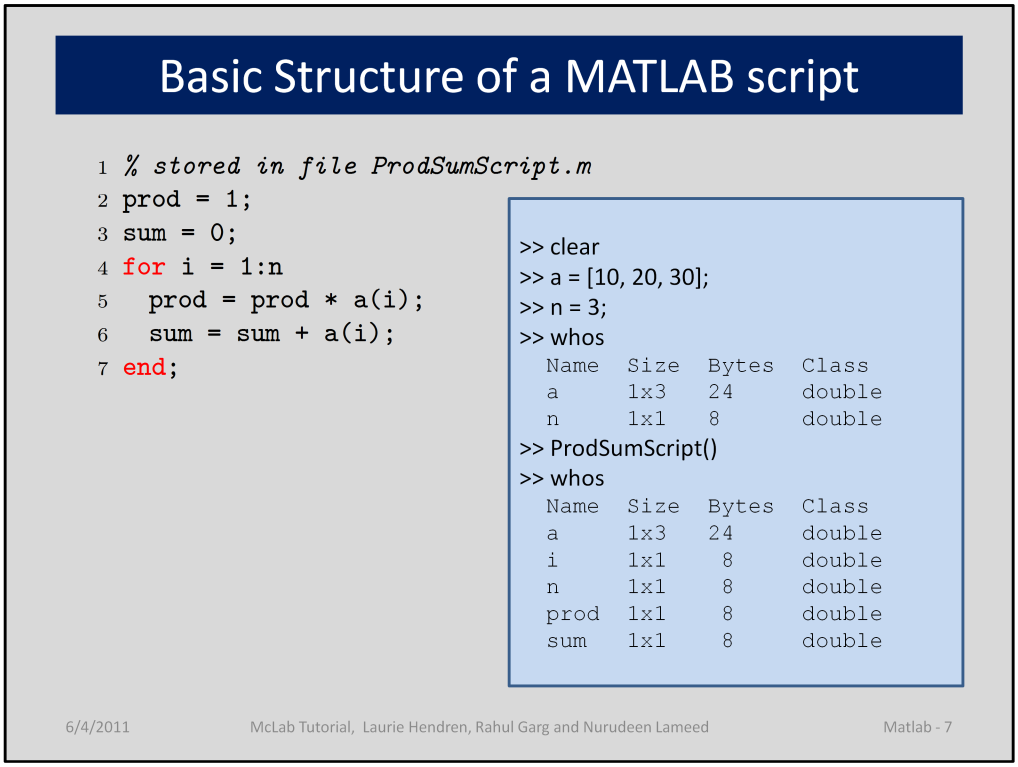


Although MATLAB programmers tend to put only one function in a file, there are some mechanisms for collecting together related functions in the same file.

The first function is called the primary function, and this one should have the same name as the file. This is the only function which is visible outside of the file.

Subsequent functions, defined after the primary function are called sub-functions. Sub-functions are only visible to other functions in the same file.

Function definitions can also be nested, and these follow the expected scoping rules. The only non-obvious point is determining which function "declares" a local variable. Effectively it is the innermost function which contains a parameter or definition of that variable.



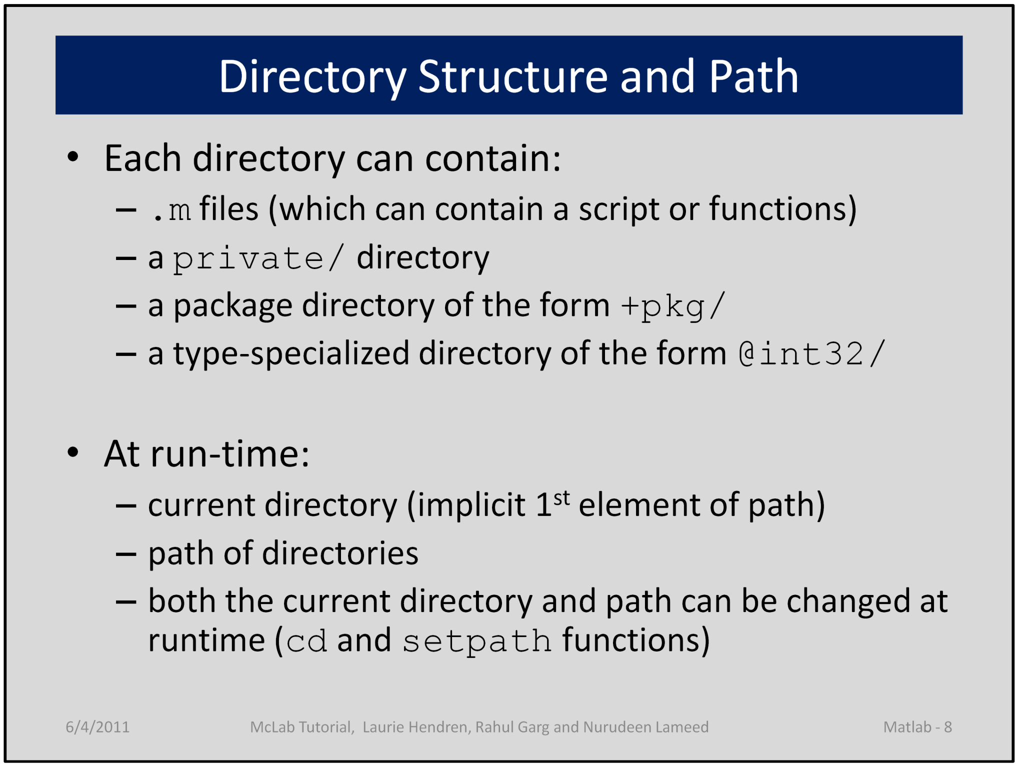
Now that we understand functions, let's look at something a little less structured, scripts ....

1st click: If a script is stored in a file foo.m, then it is called by "foo" or "foo()"

A script is neither a zero-argument function, nor a macro, but something in between.

Scripts use the workspace of their caller, which could be the read-eval-print loop, or the last-called function.

2nd click: let's look at an example of calling a script from the read-eval-print loop. First we have to ensure that the appropriate variables are defined (statements 1 through 3). Note that "whos" is a built-in function that displays the current workspace. We then call ProdSumScript, and then look in the workspace again, where we see that prod and sum have been defined.



MATLAB programmers tend to accumulate lots of functions in their current directory. However, there are mechanisms for grouping functions together.
• First, you can put them in a /private directory. These functions will be visible to functions in the outer directory.
• Second you can create directories starting with "+" which correspond to packages. Such functions must be called using pkg.f(). You can have sub-packages as well.
• Third, you can have type-specialized directories, which start with "@". These will be called when the first argument matches the type of the directory name.

At run-time a function is looked up first in the current directory, and then if not found the directories along the path are searched. Note that both the current directory and the path can be changed at run-time.

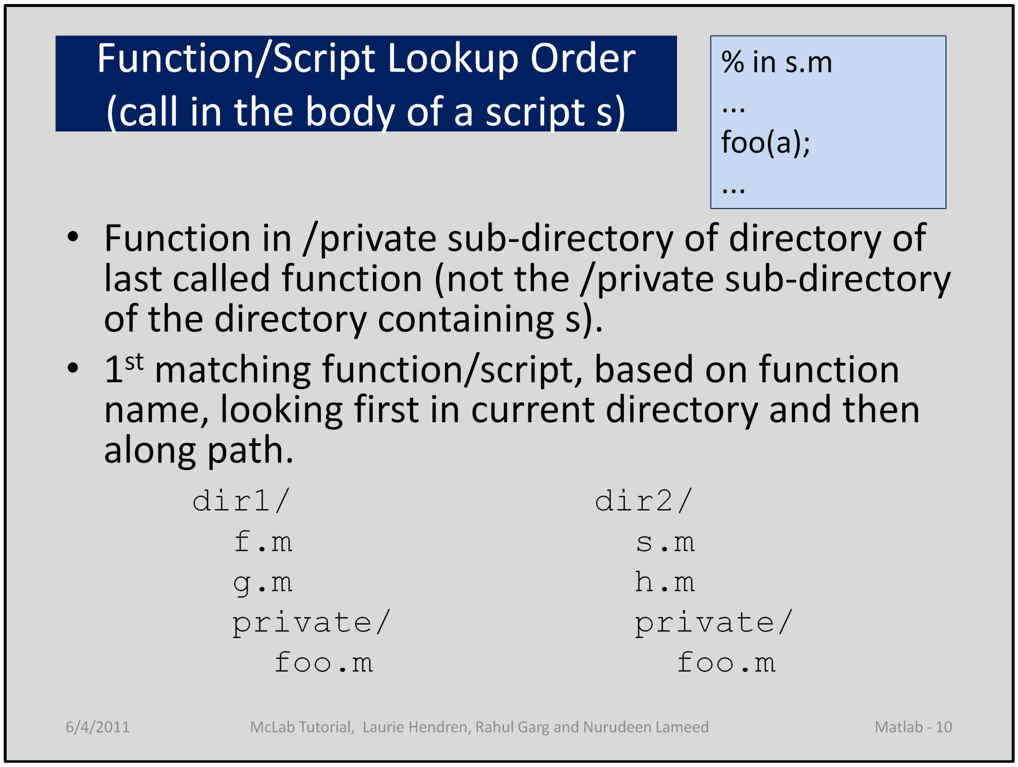
Function/Script Lookup Order (call in the body of a function f)

```
function f
  ...
  foo(a);
  ...
end
```

- Nested function (in scope of f)
- Sub-function (in same file as f)
- Function in /private sub-directory of directory containing f.
- 1st matching function, based on function name and type of first argument, looking in type-specialized directories, looking first in current directory and then along path.
- 1st matching function/script, based on function name only, looking first in current directory and then along path.

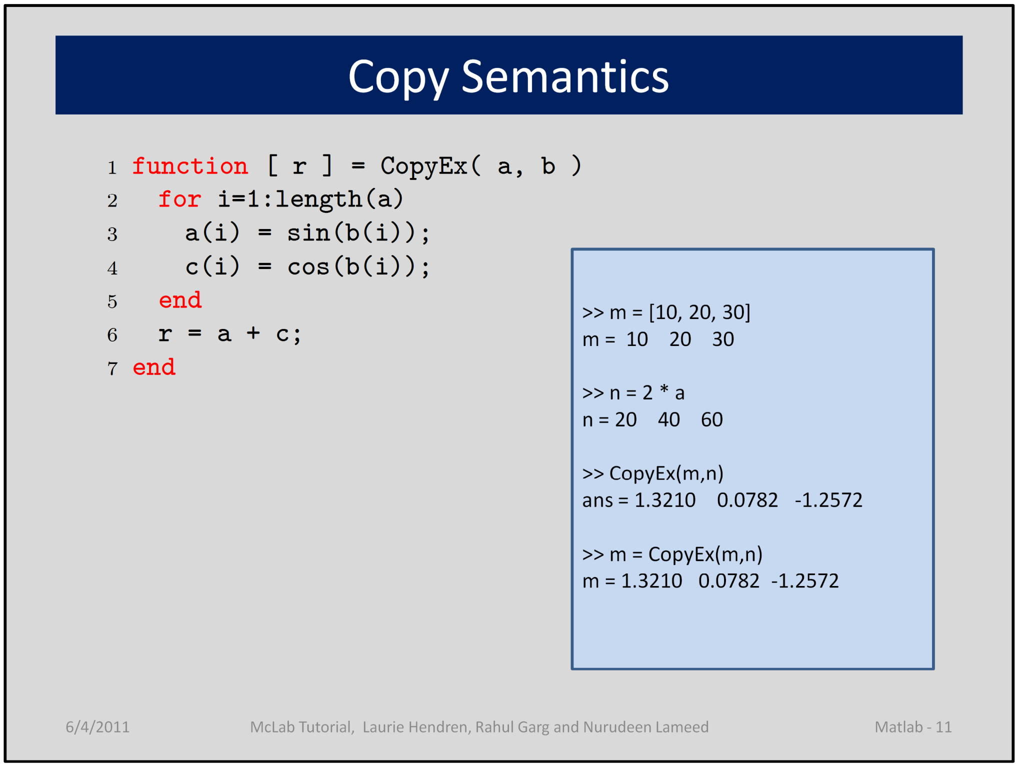
Let's look at the rules for looking up a function. They are as listed on the slide. Note that a nested, sub-function or private function takes precedence over a type-specialized function. Hence, if you want to make a function type-specialized it must be moved out a file or private directory.

In the example, for the call of foo, first look in the body of f, then in the file of f, then in the /private directory of the directory containing the file of f. If not found yet, then look along the path for a type-specialized foo that matches the run-time type of "a", and then if not found, look along the path for a function with name "foo".



Now, what about a looking up a call which is the body of a script? This is not equivalent to first macro-expanding the call. It is also different than the lookup call for functions.

•If f.m is a function definition containing a call to foo, foo will be found in dir1/private/foo.m.
• If s.m is a script definition containing a call to foo, foo will not necessarily be found in dir2/private/foo.m. Rather, it depends on the directory of the last function called. For example, if g.m called s, then foo will be dir1/private/foo.m. If h.m called s, then foo will be dir2/private/foo.m.

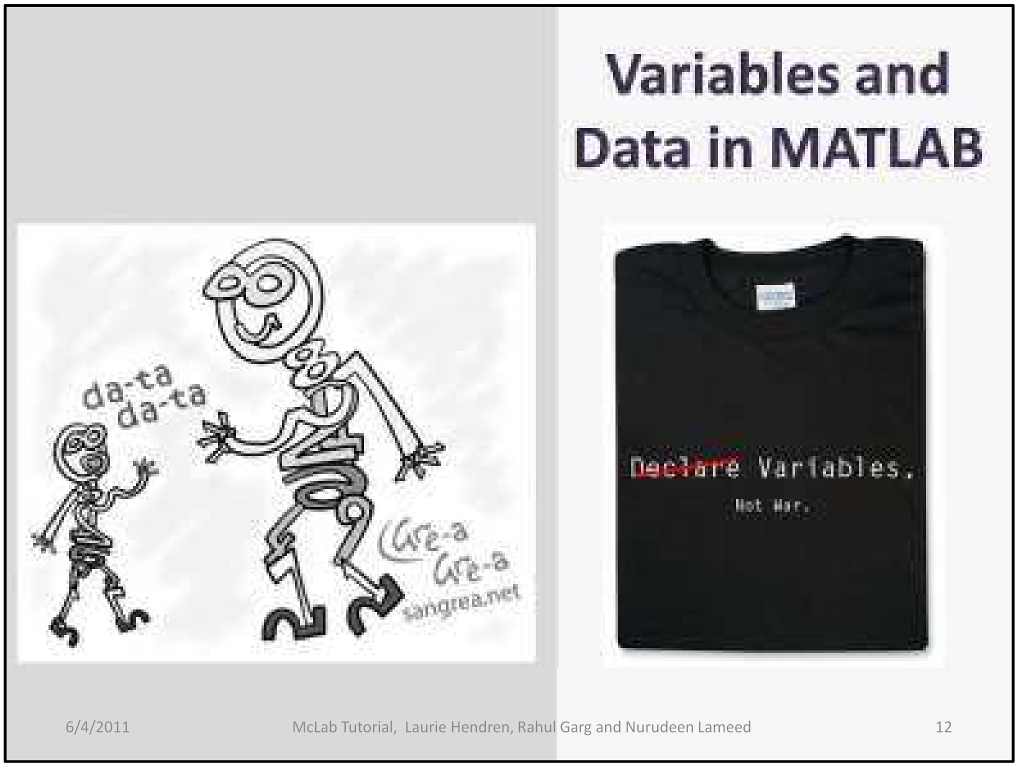


The semantics of MATLAB is call and return by value. Hence when a function call is made a copy of the input arguments are made, and then the function returns a copy of the return parameters are made. Similarly, statements of the form a = b mean that a should be a new copy.

In an implementation of MATLAB with reference counting (such as MathWorks' MATLAB and Octave), the copying is actually done lazily. At the time of parameter passing/returning or array assignments, only a reference is created, and the reference count of the pointed-to array is incremented. Then, upon any update to an array, first the reference count is checked. If the reference count is greater than 1, then a fresh copy is created at this point.

McVM uses a different approach. Since McVM supports a garbage-collected system, rather than reference counted, we use static analysis to determine when copies are needed, and the best place to insert such copies. This is reported in the paper "Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler", published in CC 2011.

Also note that fresh copies of arrays may need to be allocated if an element is assigned outside of the current range. Thus, in CopyEx, line 4, on each iteration of the for loop it will cause a fresh copy to be created, which is one element larger.



Ok, now we understand functions.... what about variables in MATLAB.

We already know they are not explicitly declared.

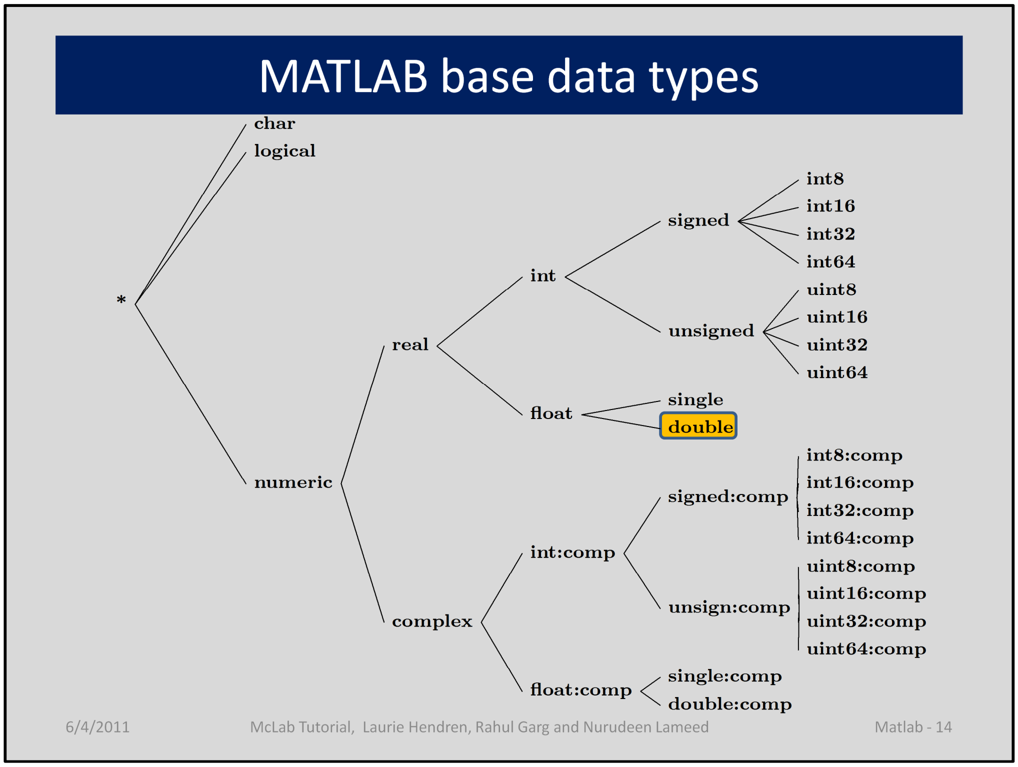
Examples of base types

```
>> clear
>> a = [10, 20, 30]
a = 10   20   30

>> b = int32(a)
b = 10   20   30

>> c = isinteger(b)
c = 1

>> d = complex(int32(4),int32(3))
d = 4 + 3i
```

```
>> whos
Name   Size  Bytes   Class    Attributes
  a     1x3   24     double
  b     1x3   12     int32
  c     1x1    1     logical
  d     1x1    8     int32     complex

>> isinteger(c)
ans = 0
>> isnumeric(a)
ans = 1
>> isnumeric(c)
ans = 0
>> isreal(d)
ans = 0
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 13

Let's first look at the base types.
If the programmer doesn't say anything different, then the base type is double – even if syntactically it looks like an integer.
To create an integer, one has to explicitly convert it, using a library function like "int32".
There are a number of built-in functions for testing the type, for example "isinteger", which return a variable with type "logical", although when printed out, they also look like integers.
There are also complex values, which have two components. These components need not be represented as doubles.



Here is our organization of the base types.

Data Conversions

- double + double → double
- single + double → double
- double:complex + double → double:complex
- int32 + double → int32

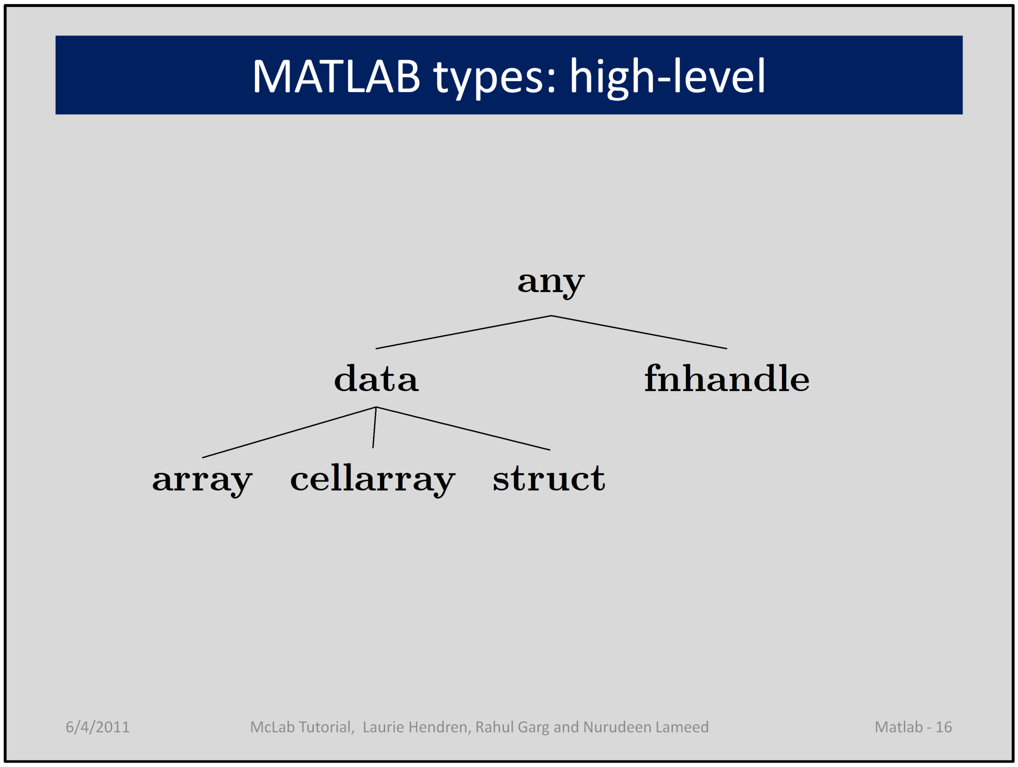
- logical + double → error, not allowed
- int16 + int32 → error, not allowed
- int32:complex + int32:complex → error, not defined

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 15

Operations dynamically check the type of their arguments and then determine the type of their return value. Not all of these conversions are as you might expect.

The first three bullet points are as you would expect, but the fourth one shows that adding an int32 to a double results in an int32.

Although doubles can be combined with other types quite easily, other combinations are not allowed, as illustrated by the final three bullet points.



Now let's look at the high-level data types. A variable can be either data or a function handle. If it is data, then it could be an array, which is homogenous, a cellarray which is effectively an array of cells, where each cell can contain a different type, or it can be a struct with named fields.

Cell array and struct example

```
>> students = {'Nurudeen', 'Rahul', 'Jesse'}
students =  'Nurudeen'   'Rahul'   'Jesse'

>> cell = students(1)
cell =   'Nurudeen'

>> contents = students{1}
contents =Nurudeen

>> whos
  Name          Size      Bytes  Class
  cell           1          128    cell
  contents      1x8          16    char
  students      1x3         372    cell
```

```
>> s = struct('name', 'Laurie',
        'student', students)
s = 1x3 struct array with fields:
      name
      student

>> a = s(1)
a = name: 'Laurie'
    student: 'Nurudeen'

>> a.age = 21
a = name: 'Laurie'
    students: 'Nurudeen'
    age:  21
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 17

Cell arrays can contain any type of data. Cell arrays are created using the { } syntax, rather than [ ] for normal arrays. Indexing into a cell array is done using x(i) to get the i'th cell, and x{i} to get the contents of the i'th cell.

Structs are created using the struct function, as shown at the top of the 2nd column. Note that in this example, since students has shape 1x3 the created struct is a 1x3 struct array with each struct containing a name field with 'Laurie' and the i'th struct containing the i'th element of students.

If we take out the first struct, and assign to a, then we have a single struct. We can add more fields to this struct by assigning to a field which doesn't yet exist, for example a.age = 21 causes a new field to be added.

Local variables

- Variables are not explicitly declared.
- Local variables are allocated in the current workspace.
- All input and output parameters are local.
- Local variables are allocated upon their first definition or via a load statement.
  - `x = ...`
  - `x(i) = ...`
  - `load ('f.mat', 'x')`
- Local variables can hold data with different types at different places in a function/script.

Variables are by default local. They are implicitly declared through assignments, or through load statements. Variables can hold different types at different places in the body of a function/script.

Global and Persistent Variables

- Variables can be declared to be global.
  - `global x;`
- Persistent declarations are allowed within function bodies only (not allowed in scripts or read-eval-print loop).
  - `persistent y;`
- A persistent or global declaration of x should cover all defs and uses of x in the body of the function/script.

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 19

It is possible to explicitly declare a variable to be global or persistent.

Variable Workspaces

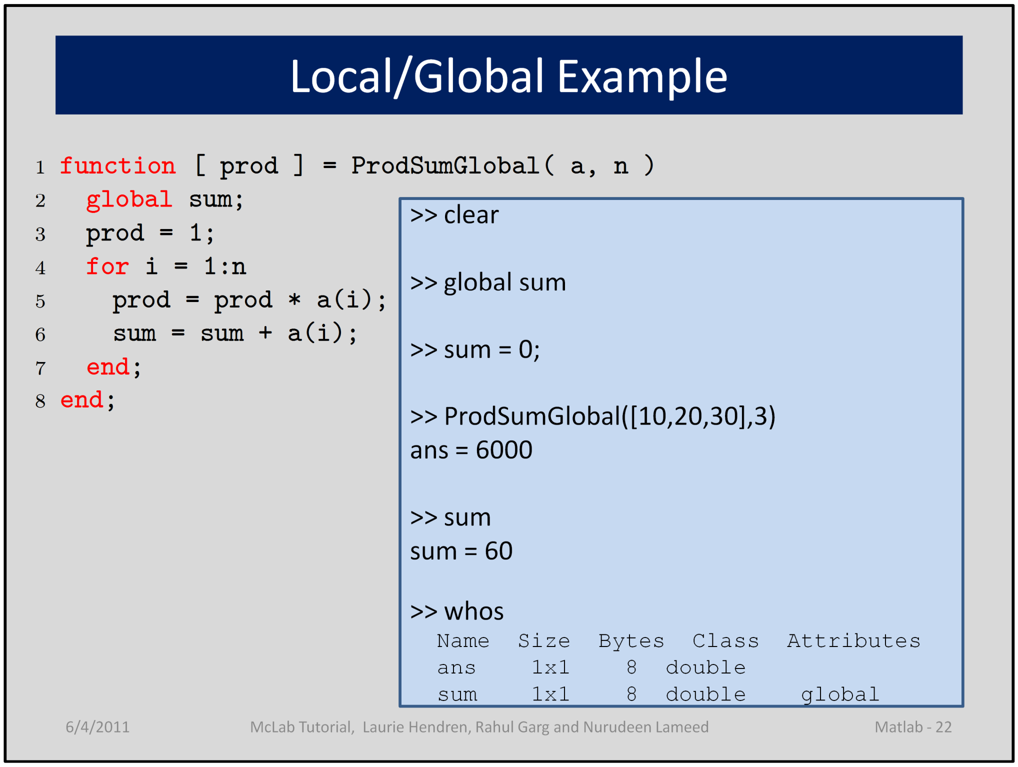
- There is a workspace for global and persistent variables.
- There is a workspace associated with the read-eval-print loop.
- Each function call creates a new workspace (stack frame).
- A script uses the workspace of its caller (either a function workspace or the read-eval-print workspace).

We can think of the environment as being a collection of workspace, one for globals and persistents, one for the read-eval-print-loop and one for each function call.

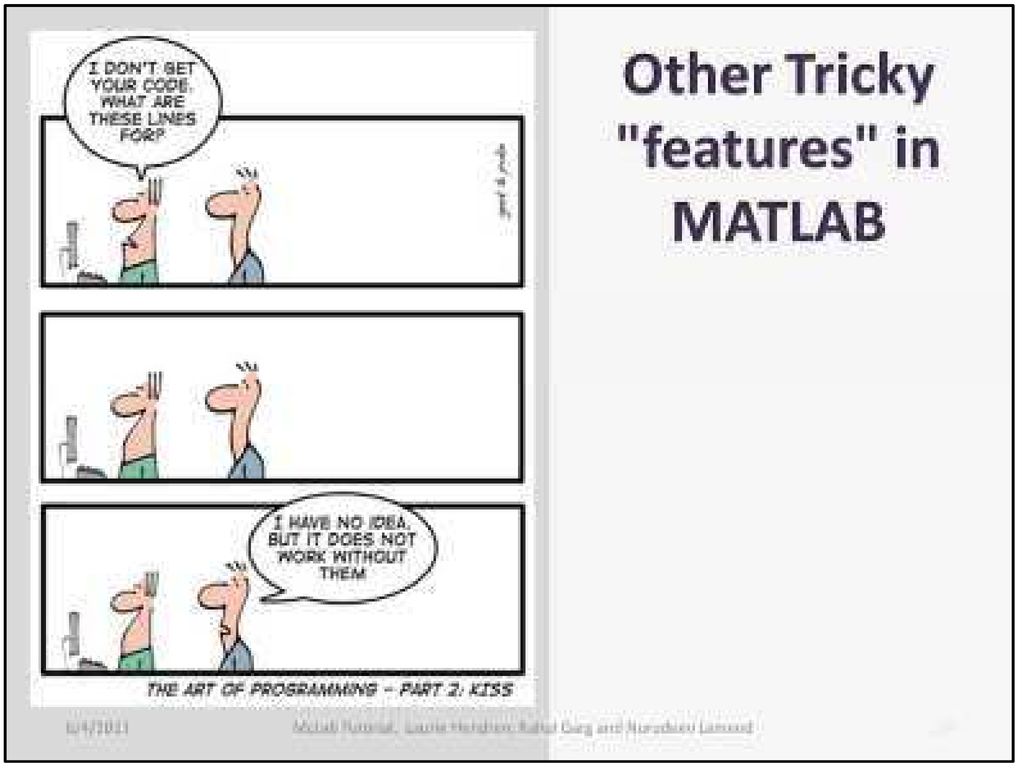
Variable Lookup

- If the variable has been declared global or persistent in the function body, look it up in the global/persistent workspace.
- Otherwise, lookup in the current workspace (either the read-eval-print workspace or the top-most function call workspace).
- For nested functions, use the standard scoping mechanisms.

How are variables looked up? If global/persistent, look in the global/persistent workspace, otherwise lookup in the current workspace. If not found, then may trigger a run-time error.



1st click: Let's look at a variation of our example where sum is a global variable instead of an input parameter.
2nd click: In the calling context we must also declare sum to be global and give it an intitial value. The default value is the the empty array, so in this case we initialize it to 0.
After calling the function, the global now has accumulated the sum of a. We can see that globals have a global attribute, when displayed with "whos".



There are some tricky and non-obvious features in MATLAB.

Looking up an identifier

Old style general lookup - interpreter

- First lookup as a variable.
- If a variable not found, then look up as a function.

MATLAB 7 lookup - JIT

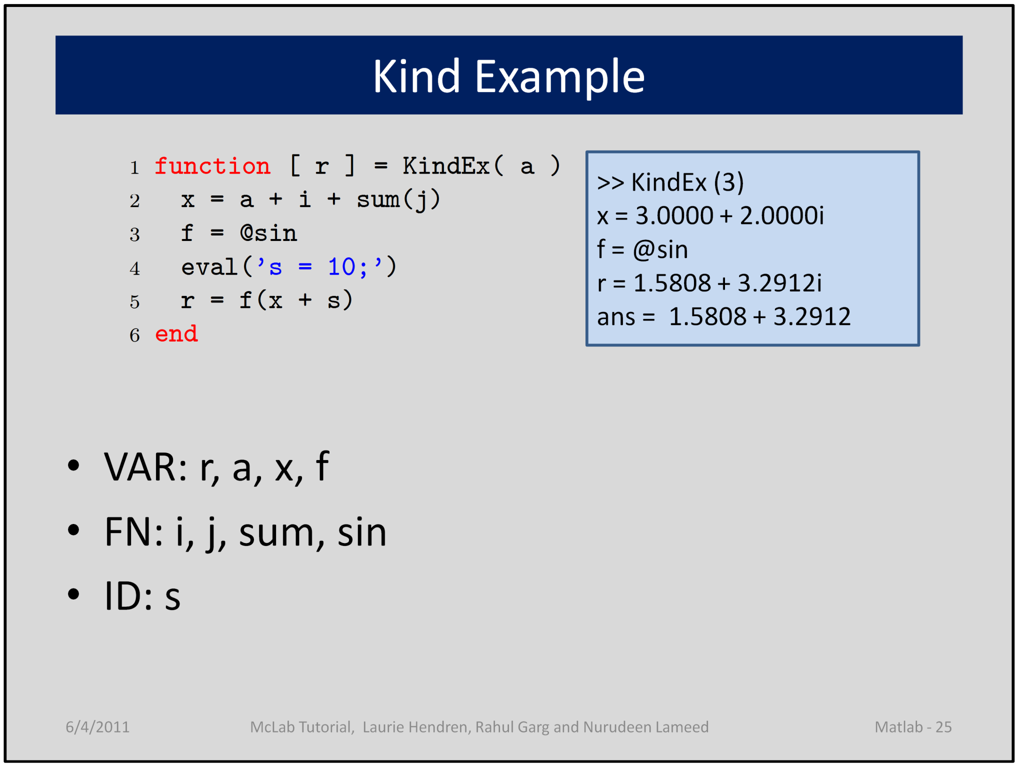
- When function/script first loaded, assign a "kind" to each identifier. VAR – only lookup as a variable, FN – only lookup as a function, ID – use the old style general lookup.

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 24

There are two styles of looking up an identifier, the old style interpreter based lookup, and a new lookup implemented in MATLAB 7, which has a JIT. This change in lookup has effectively changed the semantics of MATLAB, and so all tools that handle modern MATLAB must implement the new semantics.

In the new semantics, at first load time each identifier is assigned a kind. This is, presumably, to allow for more efficient code generation. The kinds are VAR, FN and ID.

We cannot find any documentation on this, but have written a paper on this topic, submitted to OOPSLA.



Let's take a quick look at the kind assignment algorithm. Effectively it must visit all statements in the body of the function and assign a kind. The algorithm implemented by MATHWORKs is neither flow-sensitive nor flow-insensitive, but traversal-sensitive. In this example, "a" and "r" are parameters, so they are variables. "x" and "f" are assigned-to, so they are variables. "sin" has its handle taken, so it is a FN. "i", "j", "sum" are also FN because: (a) they are not VAR, and (b) they can be found when looked in the current fn/file/directory/path. Identifier "s" is not directly definded, nor can it be found upon a function lookup, so its kind is left as ID, and a fully dynamic lookup will be used at runtime. In this case when you run the program, "s" will be found in the workspace.

2nd click: trace of running, note that the statements in the body of KindEx are not terminated by ";", so the results of the statements are echoed at run-time.

Irritating Front-end "Features"

- keyword `end` not always required at the end of a function (often missing in files with only one function).

- command syntax
  - `length('x')` or `length x`
  - `cd('mydirname')` or `cd mydirname`

- arrays can be defined with or without commas:
  [10, 20, 30] or [10 20 30]

- sometimes newlines have meaning:
  - a = [10 20 30
    40 50 60]; // defines a 2x3 matrix
  - a = [10 20 30 40 50 60]; // defines a 1x6 matrix
  - a = [10 20 30;
    40 50 60]; // defines a 2x3 matrix
  - a = [10 20 30; 40 50 60]; // defines a 2x3 matrix

There are quite a few irritating issues with the MATLAB syntax that are hard to handle with standard parsing tools.

"Evil" Dynamic Features

- not all input arguments required

```
1 function [ prod, sum ] = ProdSumNargs( a, n )
2   if nargin == 1 n = 1; end;
3   ...
4 end
```

- do not need to use all output arguments
- eval, evalin, assignin
- cd, addpath
- load

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Matlab - 27

There are also several potentially evil dynamic features.

First, when a function is called, not all arguments need to be provided. In the body of the function one can check to see how many arguments were provided, and then assign a default value to the missing ones (as in line 2 of ProdSumNargs).

Similarly a call to a function need not capture all of the output arguments.

There are several kinds of eval, including the ordinary one, evalin – which is used to eval an expression is a different context (such as the calling functions context) and assignin which is used to assign to a variable in the calling context.

Cd and addpath can be used to dynamically change the current directory or modify the path, which causes a change in function lookup.

Load can be used to load stored variables from a file, and so may cause new variables to be created in the current workspace.

McLab Tutorial
www.sable.mcgill.ca/mclab

Part 3 – McLab Frontend

- Frontend organization
- Introduction to Beaver
- Introduction to JastAdd

McLab Frontend

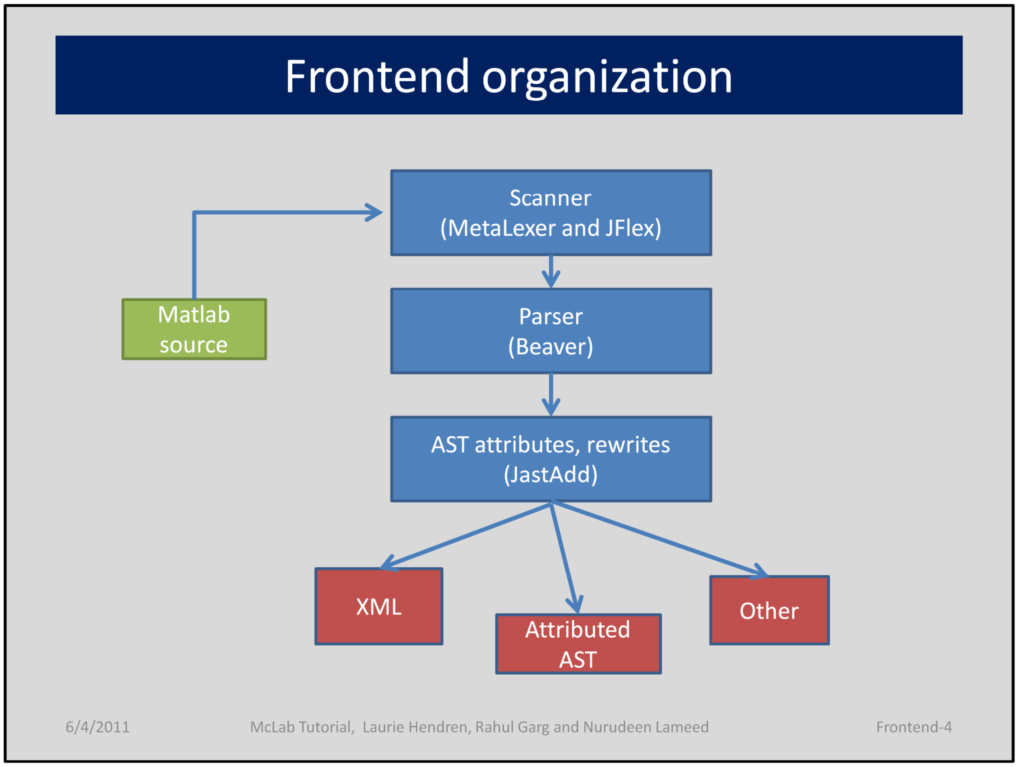
- Tools to parse MATLAB-type languages
  - Quickly experiment with language extensions
  - Tested on a lot of real-world Matlab code
- Parser generates ASTs
- Some tools for computing attributes of ASTs
- A number of static analyses and utilities
  - Example: Printing XML representation of AST

Tools used

- Written in Java (JDK 6)
- MetaLexer and JFlex for scanner
- Beaver parser generator
- JastAdd “compiler-generator” for computations of AST attributes
- Ant based builds
- We typically use Eclipse for development
  - Or Vim ☺

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-3

Look! Notes!

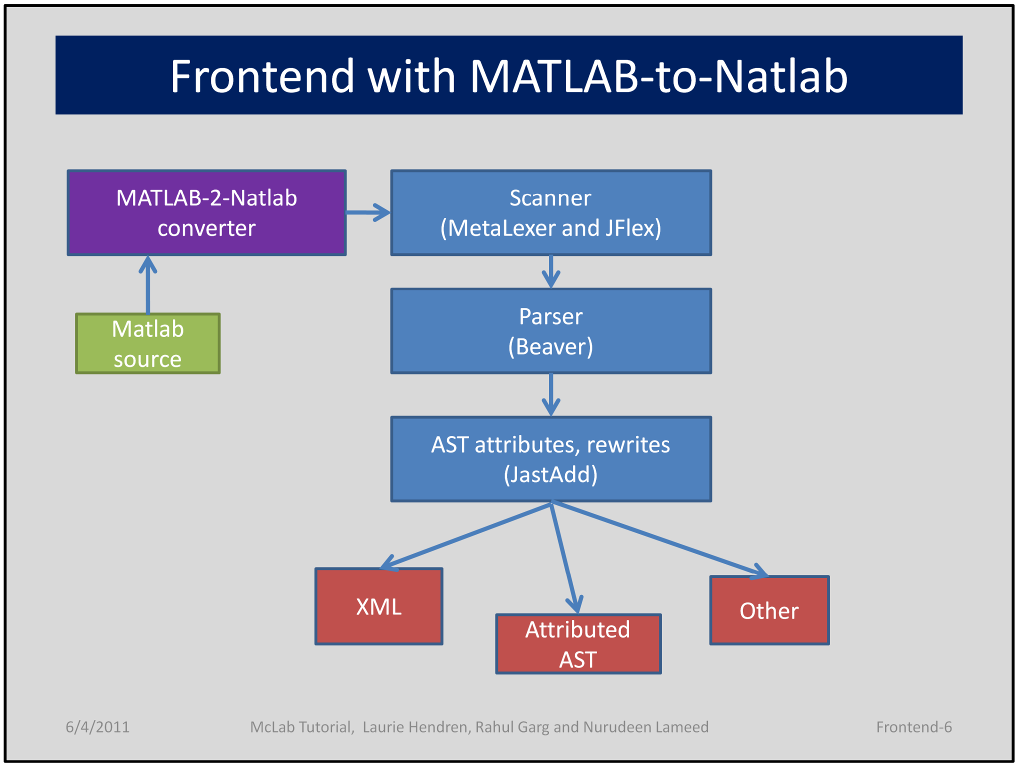

Frontend organization
Scanner
(MetaLexer and JFlex)
Matlab
source
Parser
(Beaver)
AST attributes, rewrites
(JastAdd)
XML
Attributed
AST
Other
6/4/2011
McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed
Frontend-4

Natlab

- Natlab is a clean subset of MATLAB
  - Not a trivial subset though
  - Covers a lot of “sane” MATLAB code
- MATLAB to Natlab translation tool available
  - Written using ANTLR
  - Outside the scope of this tutorial
- Forms the basis of much of our semantics and static analysis research

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-5

Derivatives such as AspectMatlab use the work done in Natlab.


Frontend with MATLAB-to-Natlab
MATLAB-2-Natlab converter
Matlab source
Scanner (MetaLexer and JFlex)
Parser (Beaver)
AST attributes, rewrites (JastAdd)
XML
Attributed AST
Other
6/4/2011
McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed
Frontend-6

How is Natlab organized?

- Scanner specifications
  - src/metalexer/shared_keywords.mlc
- Grammar files
  - src/parser/natlab.parser
- AST computations based on JastAdd
  - src/natlab.ast
  - src/*jadd, src/*jrag
- Other Java files
  - src/*java

MetaLexer

- A system for writing extensible scanner specifications
- Scanner specifications can be modularized, reused and extended
- Generates JFlex code
  – Which then generates Java code for the lexer/scanner
- Syntax is similar to most other lexers
- Reference: "MetaLexer: A Modular Lexical Specification Language. Andrew Casey, Laurie Hendren" by Casey, Hendren at AOSD 2011.

If you already know Beaver and JastAdd…

Then take a break.
Play Angry Birds.
Or Fruit Ninja.

Frontend-9

Beaver

- Beaver is a LALR parser generator
- Familiar syntax (EBNF based)
- Allows embedding of Java code for semantic actions
- Usage in Natlab: Simply generate appropriate AST node as semantic action

Beaver Example

```
Stmt stmt =
        expr.e {: return new ExprStmt(e); :}
|       BREAK {: return new BreakStmt(); :}
|       FOR   for_assign.a  stmt_seq.s END
                     {: return new ForStmt(a,s); :}
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-11

Example is a simplified grammar

Beaver Example

Java type

```
Stmt stmt =
        expr.e {: return new ExprStmt(e); :}
|       BREAK {: return new BreakStmt(); :}
|       FOR   for_assign.a  stmt_seq.s END
                        {: return new ForStmt(a,s); :}
```


The Java types must be declared/defined/imported by the programmer.

Beaver Example

Node name in grammar

```
Stmt stmt =
        expr.e {: return new ExprStmt(e); :}
|       BREAK {: return new BreakStmt(); :}
|       FOR   for_assign.a  stmt_seq.s END
                        {: return new ForStmt(a,s); :}
```

Beaver Example

Identifier for node

```
Stmt s
        expr.e {: return new ExprStmt(e); :}
|       BREAK {: return new BreakStmt(); :}
|       FOR   for_assign.a  stmt_seq.s END
                        {: return new ForStmt(a,s); :}
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-14

The name given to a node can then be used inside the semantic action.

Beaver Example

Java code for semantic action

```
Stmt stmt =
        expr.e {: return new ExprStmt(e); :}
|       BREAK {: return new BreakStmt(); :}
|       FOR   for_assign.a  stmt_seq.s END
                        {: return new ForStmt(a,s); :}
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-15

JastAdd: Motivation

- You have an AST
- Each AST node type represented by a class
- Want to compute attributes of the AST
  - Example: String representation of a node
- Attributes might be either:
  - Inherited from parents
  - Synthesized from children

JastAdd

- JastAdd is a system for specifying:
  - Each attribute computation specified as an aspect
  - Attributes can be inherited or synthesized
  - Can also rewrite trees
  - Declarative philosophy
  - Java-like syntax with added keywords
- Generates Java code
- Based upon "Reference attribute grammars"

How does everything fit?

- JastAdd requires two types of files:
  - .ast file which specifies an AST grammar
  - .jrag/.jadd files which specify attribute computations
- For each node type specified in AST grammar:
  - JastAdd generates a class derived from ASTNode
- For each aspect:
  - JastAdd adds a method to the relevant node classes

JastAdd AST File example

```
abstract BinaryExpr: Expr ::=
        LHS:Expr RHS:Expr
PlusExpr: BinaryExpr;
MinusExpr: BinaryExpr;
MTimesExpr: BinaryExpr;
```

JastAdd XML generation aspect

```
aspect AST2XML{
..
eq BinaryExpr.getXML(Document d, Element e){
        Element v = d.getElement(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
...
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Frontend-20

Code has been simplified to suit the purposes of the tutorial. Actual code will do a little more bookkeeping of line numbers etc.

Aspect declaration

```
aspect AST2XML{
..
eq BinaryExpr.getXML(Document d, Element e){
        Element v = d.getElement(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
...
```


Code has been simplified to suit the purposes of the tutorial. Actual code will do a little more bookkeeping of line numbers etc.

```
aspect AST2XML{
```



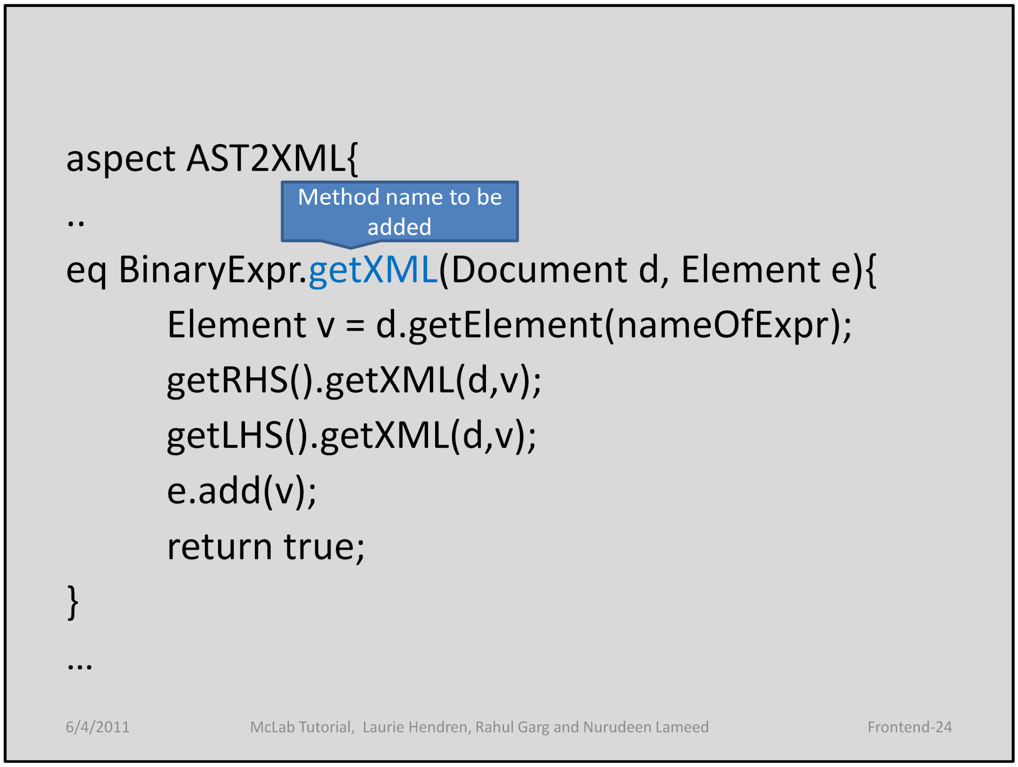
```
eq BinaryExpr.getXML(Document d, Element e){
        Element v = d.getElement(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
...
```


Code has been simplified to suit the purposes of the tutorial. Actual code will do a little more bookkeeping of line numbers etc.

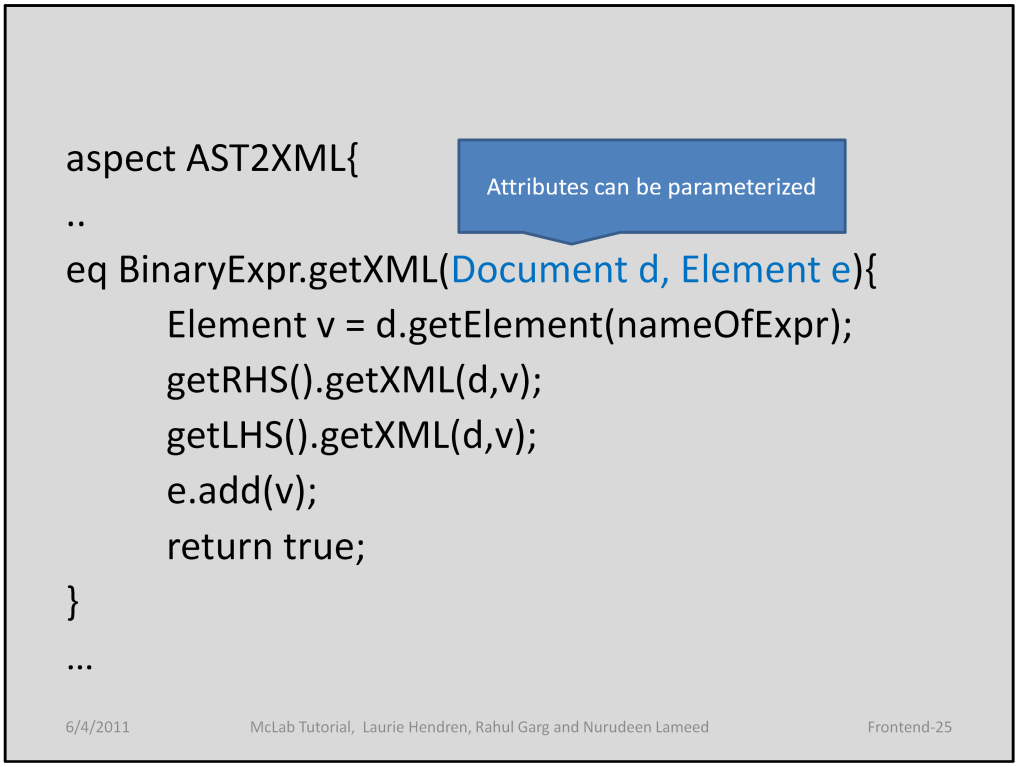
Add to this AST class

```
aspect AST2XML{
..
eq BinaryExpr.getXML(Document d, Element e){
        Element v = d.getElement(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
...
```


Code has been simplified to suit the purposes of the tutorial. Actual code will do a little more bookkeeping of line numbers etc.



Code has been simplified to suit the purposes of the tutorial. Actual code will do a little more bookkeeping of line numbers etc.

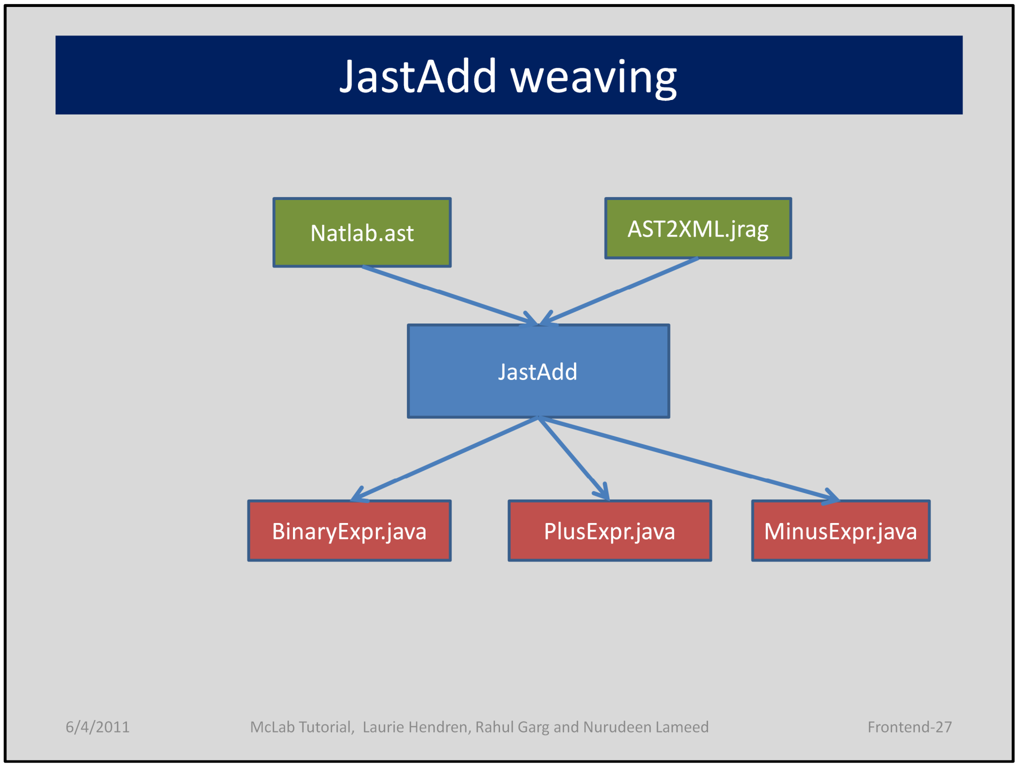


Code has been simplified to suit the purposes of the tutorial. Actual code will do a little more bookkeeping of line numbers etc.

```
aspect AST2XML{
..
eq Binar          Document d, Element e){
        Ele          Element(nameOfExpr);
        getRHS().getXML(d,v);
        getLHS().getXML(d,v);
        e.add(v);
        return true;
}
...
```

Compute for children

Code has been simplified to suit the purposes of the tutorial. Actual code will do a little more bookkeeping of line numbers etc.


JastAdd weaving
Natlab.ast
AST2XML.jrag
JastAdd
BinaryExpr.java
PlusExpr.java
MinusExpr.java
6/4/2011
McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed
Frontend-27

Overall picture recap

- Scanner converts text into a stream of tokens
- Tokens consumed by Beaver-generated parser
- Parser constructs an AST
- AST classes were generated by JastAdd
- AST classes already contain code for computing attributes as methods
- Code for computing attributes was weaved into classes by JastAdd from aspect files

Adding a node

- Let's assume you want to experiment with a new language construct:
- Example: parallel-for loop construct
  - parfor i=1:10 a(i) = f(i) end;
- How do you extend Natlab to handle this?
- You can either:
  - Choose to add to Natlab source itself
  - (Preferred) Setup a project that inherits code from Natlab source directory

Steps

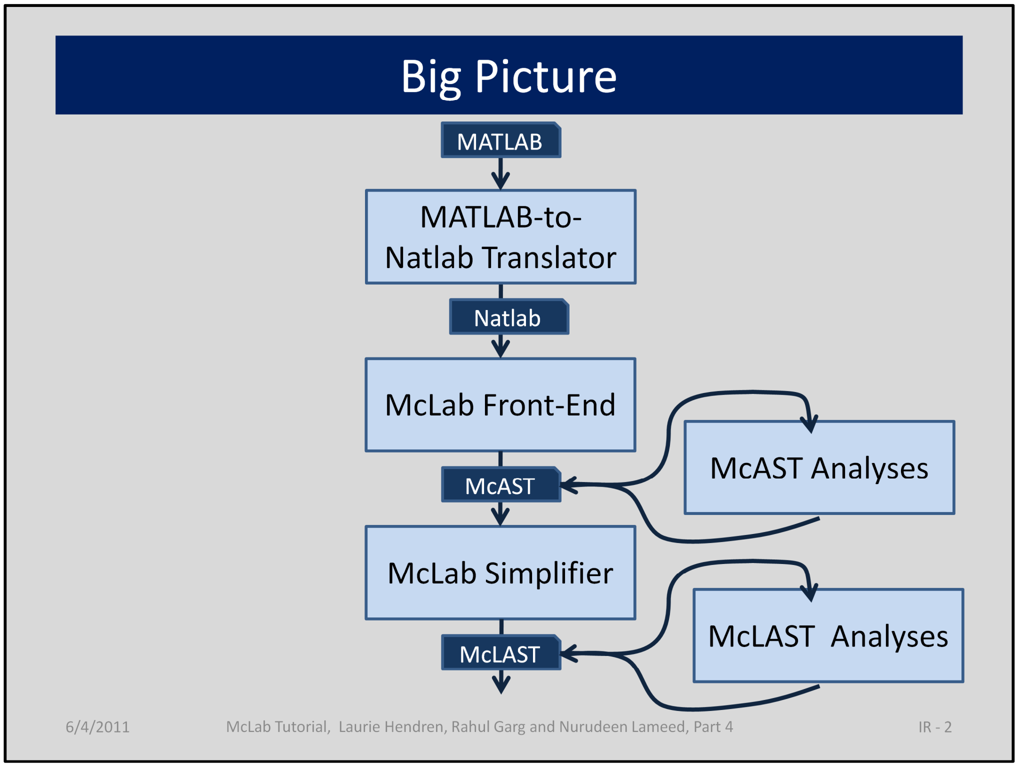
- Write the following in your project:
  - Lexer rule for “parfor”
  - Beaver grammar rule for parfor statement type
  - AST grammar rule for PforStmt
  - attributes for PforStmt according to your requirement
  - eg. getXML() for PforStmt in a JastAdd aspect
  - Buildfile that correctly passes the Natlab source files and your own source files to tools
  - Custom main method and jar entrypoints

McLab Tutorial
www.sable.mcgill.ca/mclab

Part 4 – McLab Intermediate Representations

- High-level McAST
- Lower-level McLAST
- Transforming McAST to McLAST

Now that we have explained the front-end, we need to explain our IRs. We will start with the high-level AST, McAST, which is produced by the front-end, then we will describe our lower-level IR, McLAST, which is still tree-based, but is simplified and more suitable for flow analysis. Then we will describe how we transform McAST into McLAST.



We have already shown you the top two light blue boxes in this figure. The front-end produces McAST. Now, we must do some analysis on this AST to determine the kind of each identifier (VAR, FN or ID), and then simplify the McAST yielding a lower level representation called McLAST. Various analyses can then be applied to McLAST. Both the McAST and McLAST analyses can be implemented using our flow analysis framework, which will be introduced in the next part of this tutorial.

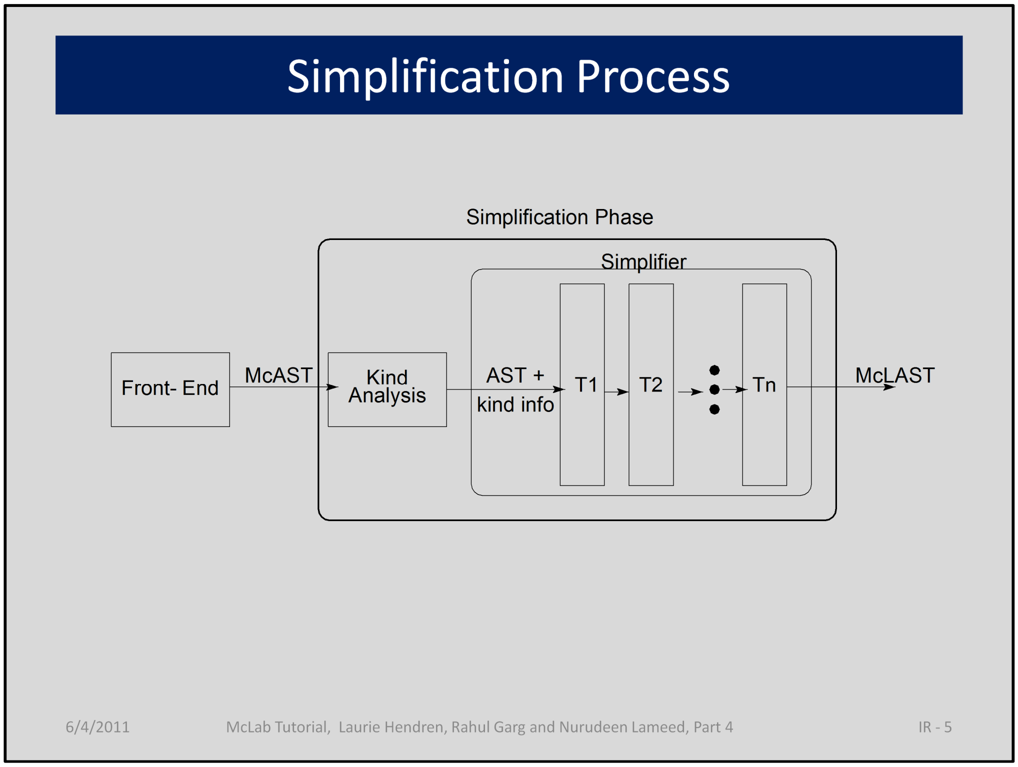
McAST

- High-level AST as produced from the front-end.
- AST is implemented via a collection of Java classes generated from the JastAdd specification file.
- Fairly complex to write a flow analysis for McAST because of:
  - arbitarly complex expressions, especially lvalues
  - ambiguous meaning of parenthesized expressions such as a(i)
  - control-flow embedded in expressions (&&, &, ||, |)
  - MATLAB-specific issues such as the "end" expression and returning multiple values.

Best source of further documentation – Chapters 3 and 4 of Jesse Doherty's M.Sc. thesis.

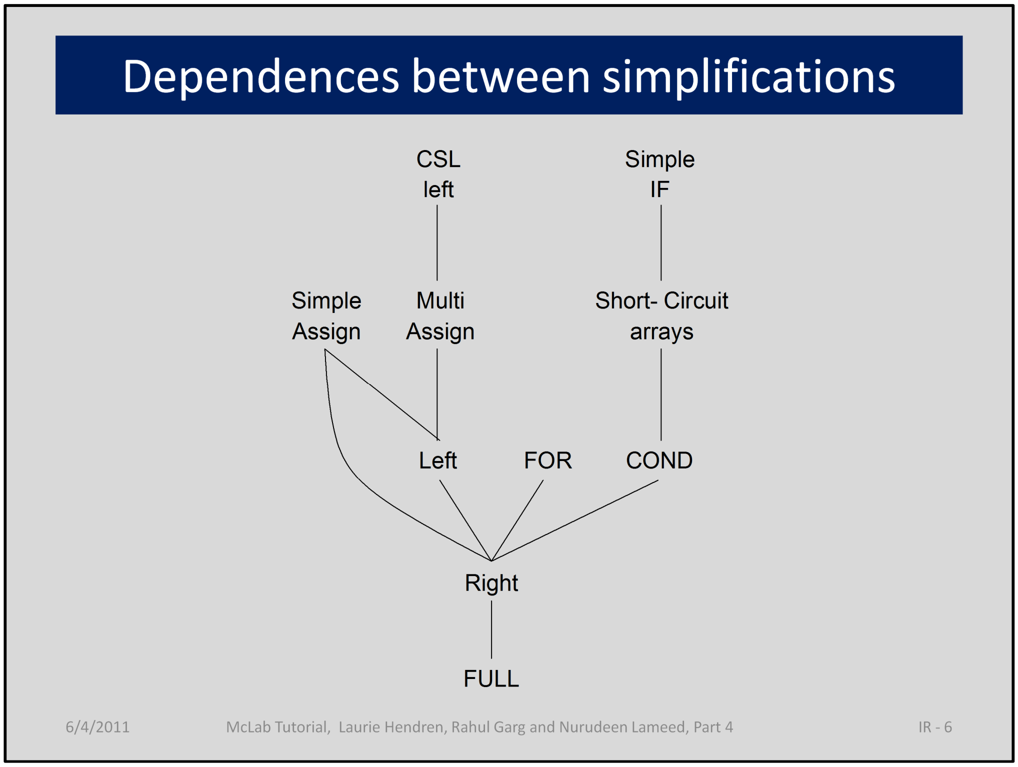
McLAST

- Lower-level AST which:
  - has simpler and explicit control-flow;
  - simplifies expressions so that each expression has a minimal amount of complexity and fewer ambiguities; and
  - handles MATLAB-specific issues such as "end" and comma-separated lists in a simple fashion.
- Provides a good platform for more complex flow analyses.

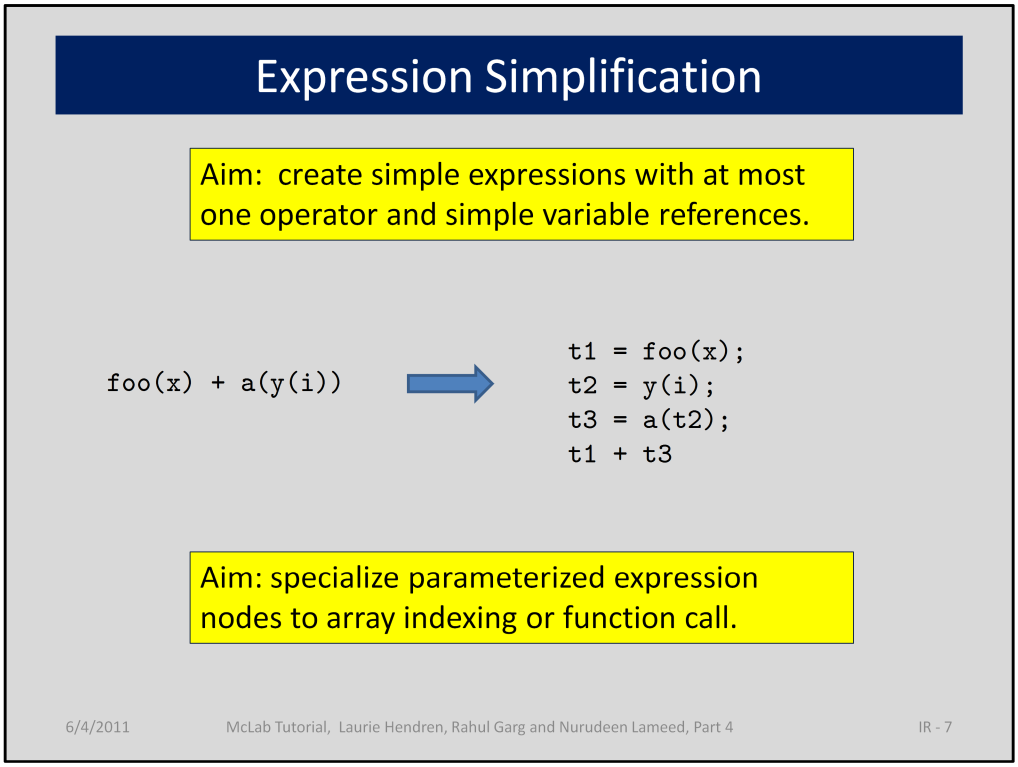


The simplificatin proceeds by getting the McAST from the front-end, and then applying the kind analysis to that AST, then given the AST and the kind analysis info a sequence of simplifying transformations are applied, finally yielding the lower-level McLAST.

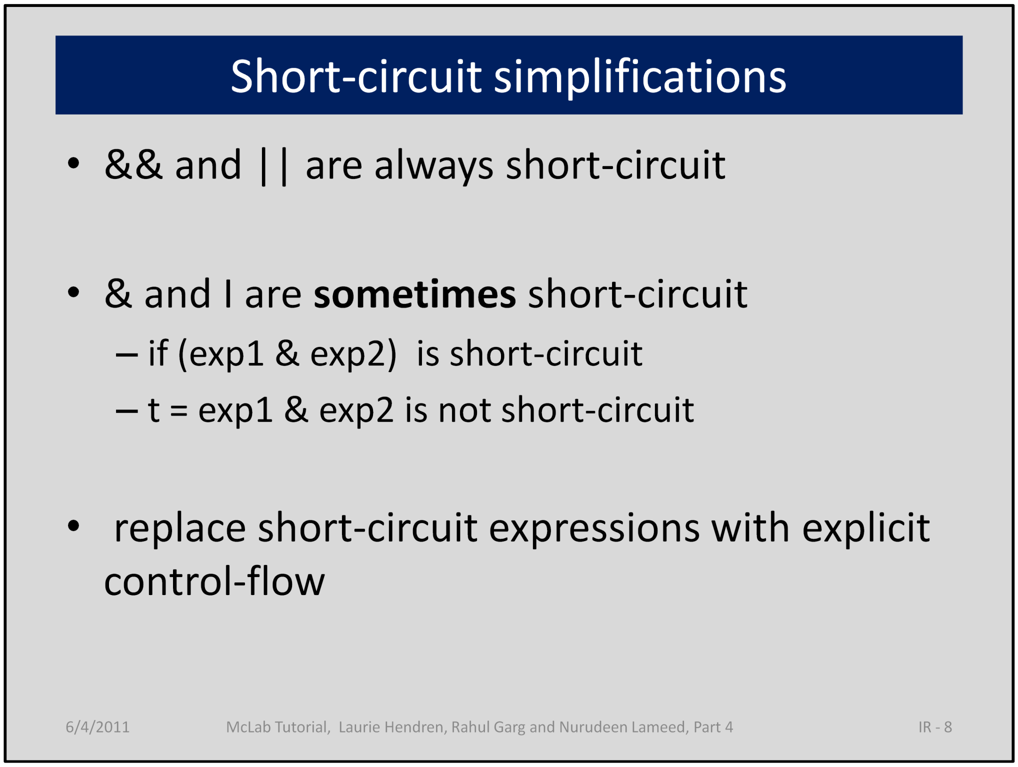
The default behaviour is that all the simplifications are run, but there may be situations where a framework user only wants to apply some of the simplifications. However, some simplifications depend on others having already been performed, so how do we deal with this?



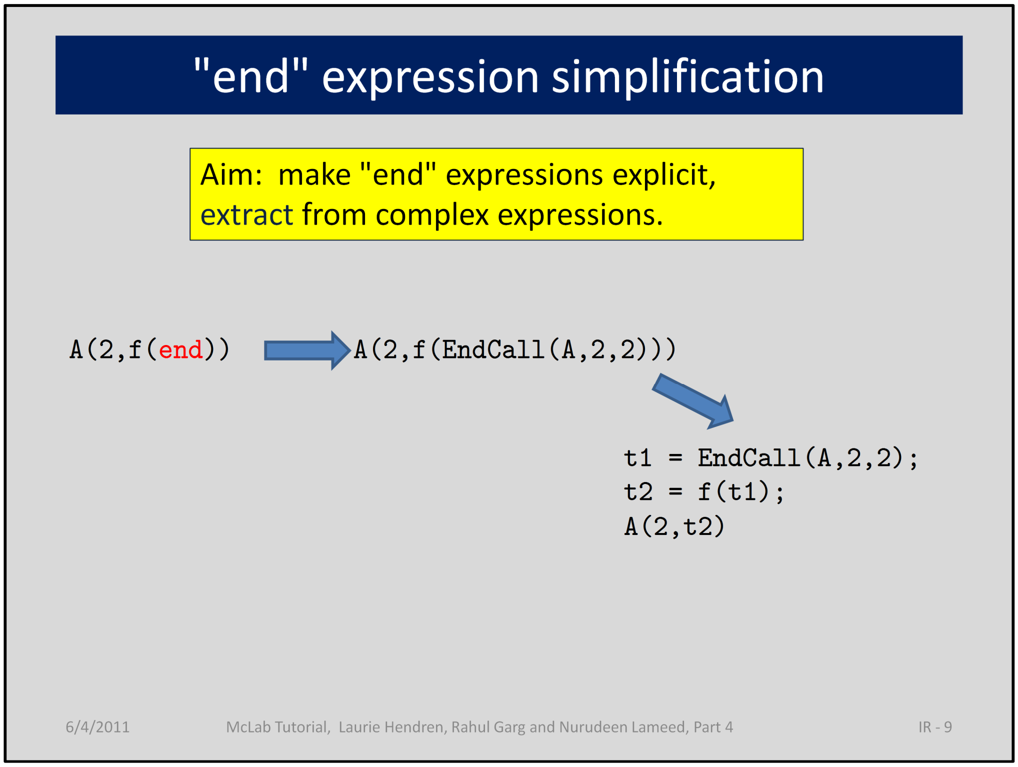
Each simplification specifies which other simplifications it depends on, giving us a DAG of dependences. Now, if only some simplifications are desired, the system will run only those simplifications, plus those that it depends on. If a user adds a new simplification to the framework, they must also specify which simplifications it depends on.



One of the main simplifications is to simplify expressions, so that each expression has at most one operation. In addition we need to resolve the meaning of expressions like a(i) which in the high-level AST are stored as parameterized expressions. In the simplification we can to encode these as either array indexing or function calls.



In order to simplify program analysis, we want to remove any control-flow from expressions. The && and || operators are always short-circuit, and so do involve control flow. Thus these are always converted to equivalent nested conditional statements. However, in MATLAB the itemwise operators & and | are also sometimes short-circuit. If the appear in the condition of an if or while they are short-circuit, otherwise they are not. Thus, only the short-circuit occurances are simplified.



MATLAB includes and "end" expression which is used to capture the last index of the closest enclosing array. So for example,
A(2,f(end)) could mean two things. If f is a variable, then it means the last index of f. If f is a function and A is a variable, then it means the last index of the 2nd dimension of A, where A is being indexed using 2 dimensions. We use the kind analysis to determine the closest enclosing variable, and then convert the end expression into and explicit EndCall.

In this new expression, EndCall(A,2,2) is the explicit end. It is specifying that the end binds to
the array A interpreted as two dimensional where the end is used in the second dimension.

L-value Simplification

Aim: create simple l-values.

```
A(a+b,2).e(foo()) = value;
```

⇒

```
t1 = a+b;
t2 = foo();
A(t1,2).e(t2) = value;
```

Note: no mechanism for taking the address of location in MATLAB. Further simplification not possible, while still remaining as valid MATLAB.

Simplifying l-values is a bit tricky in MATLAB. We want to break down the computation of l-values as much as possible However, MATLAB has no way of taking the address of variables, so we can only simplify these expressions to a limited extent.

if statement simplification

Aim: create if statements with only two control flow paths.

```
if E1
  body1();
elseif E2
  body2();
else
  body3();
end
```

```
if E1
  body1();
else
  if E2
    body2();
  else
    body3();
  end
end
```


MATLAB if statements have multiple possible elseif branches. To simplify subsequent flow analysis, we convert these into if-then-else which have at most two branches.

for loop simplification

Aim: create for loops that iterate over a variable incremented by a fixed constant.

```
for i = 1:2:n
  % BODY
end
```

```
for i = E
  % BODY
end
```

```
t1=E;
t2=size(t1);
t3=prod(t2(2:end));
for t4 = 1:t3
  i = t1(t4);
  % BODY
end
```


The semantics of a for loop are that the rhs expression of the header is evaluated to form a vector, and then the body of the loop is executed is executed over each of the elements of this vector. If it is higher-dimensional array, then it loops through the columns. When the rhs expression is very simple, generated by the colon operator, then the values of i are quite simple to reason about and will behave like an ordinary induction variable. However, when the rhs is some arbitrary other expression E, then we convert the loop to something that does use a simple colon expression in the head of the loop.

McLab Tutorial
www.sable.mcgill.ca/mclab

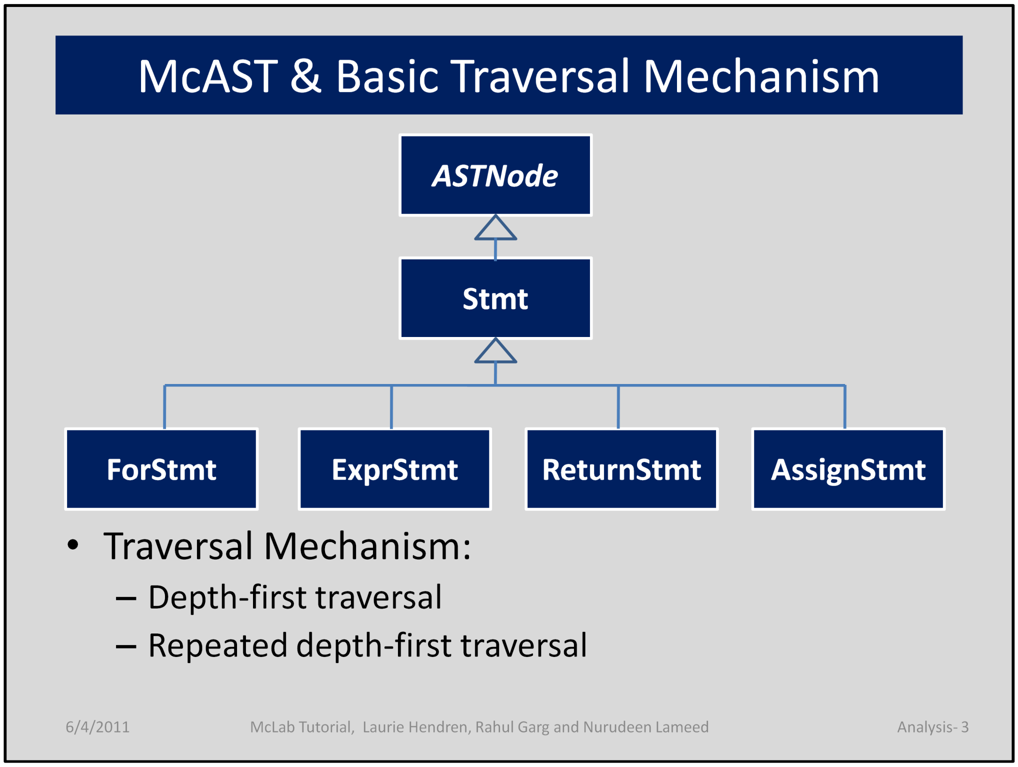
Part 5 – Introduction to the McLab Analysis Framework

- Exploring the Main Components
- Creating a Simple Analysis
- Depth-first and Structural Analyses
- Example: Reaching Definition Analysis

McLab Analysis Framework

- A simple static flow analysis framework for MATLAB-like languages
- Supports the development of intra-procedural forward and backward flow analyses
- Extensible to new language extensions
- Facilitates easy adaptation of old analyses to new language extensions
- Works with McAST and McLAST (a simplified McAST)

The Mclab analysis framework is designed to be simple to use. It is easily extensible for developing analyses for new language extensions. It is written in Java programming language



A simplified and incomplete McLab Abstract Syntax Tree; ASTNode class is the root node of McAST. With Repeated depth-first traversal, some nodes are visited repeatedly to compute a fixed point for an analysis.

Exploring the main components for developing analyses

The interface *NodeCaseHandler*

- Declares all methods for the action to be performed when a node of the AST is visited:

```
public interface NodeCaseHandler {
   void caseStmt(Stmt node);
   void caseForStmt(ForStmt node);
   void caseWhileStmt(WhileStmt node);
   …
}
```

The class *AbstractNodeCaseHandler*

```
public class AbstractNodeCaseHandler implements
   NodeCaseHandler {
   …
   void caseStmt(Stmt node) {
        caseASTNode(node);
   }
   …
}
```

- Implements the interface *NodeCaseHandler*
- Provides default behaviour for each AST node type except for the root node (*ASTNode*)

AbstractNodeCaseHandler provides default implementation for all node types except the root node. The default behaviour of a node case hander is to forward to the node case handler of its parent.

The analyze method

- Each AST node also implements the method *analyze* that performs an analysis on the node:

```
public void analyze(NodeCaseHandler handler)
    handler.caseAssignStmt(this);
}
```

Creating a simple analysis

Analysis- 8

Creating a Traversal/Analysis:

- Involves 3 simple steps:

  1. Create a concrete class by extending the class *AbstractNodeCaseHandler*
  2. Provide an implementation for *caseASTNode*
  3. Override the relevant methods of *AbstractNodeCaseHandler*

An Example: StmtCounter

- Counts the number of statements in an AST

Analysis development Steps:

1. Create a concrete class by extending the class *AbstractNodeCaseHandler*
2. Provide an implementation for *caseASTNode*
3. Override the relevant methods of *AbstractNodeCaseHandler*

An Example: StmtCounter

1. Create a concrete class by extending the class *AbstractNodeCaseHandler*

```
public class StmtCounter extends
    AbstractNodeCaseHandler {
    private int count = 0;
    … // defines other internal methods
}
```

An Example: StmtCounter --- Cont'd

2. Provide an implementation for *caseASTNode*

```
public void caseASTNode( ASTNode node){
   for(int i=0; i<node.getNumChild(); ++i) {
      node.getChild(i).analyze(this);
   }
}
```

An Example: StmtCounter --- Cont'd

3. Override the relevant methods of *AbstractNodeCaseHandler*

```
public void caseStmt(Stmt node) {
    ++count;
    caseASTNode(node);
}
```

An Example: StmtCounter --- Cont'd

```
public class StmtCounter extends AbstractNodeCaseHandler {
    private int count = 0;
    private StmtCounter() { super(); }
    public static int countStmts(ASTNode tree) {
        tree.analyze(new StmtCounter());
    }
    public void caseASTNode( ASTNode node){
        for(int i=0; i<node.getNumChild(); ++i) {
                node.getChild(i).analyze(this);}
    }
    public void caseStmt(Stmt node) {
        ++count; caseASTNode(node);
    }
}
```

Tips: Skipping Irrelevant Nodes

For many analyses, not all nodes in the AST are relevant; to skip unnecessary nodes override the handler methods for the nodes. For Example:

```
public void caseExpr(Expr node) {
    return;
}
```

Ensures that all the children of *Expr* are skipped

Analyses Types: Depth-first and Structural Analyses

Analysis-16

Flow Facts: The interface *FlowSet*

- The interface *FlowSet* provides a generic interface for common operations on flow data

```
public  interface FlowSet<D> {
  public FlowSet<D> clone();
  public void copy(FlowSet<? super D> dest);
  public void union(FlowSet<? extends D> other);
  public void intersection(FlowSet<? extends D> other);
     …
}
```

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Analysis- 17

A typical analysis computes a set of flow facts or data

The *Analysis* interface

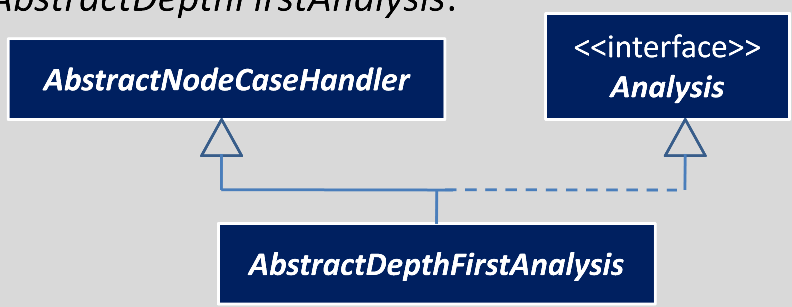
- Provides a common API for all analyses
- Declares additional methods for setting up an analysis:

```
public interface Analysis<A extends FlowSet> extends
    NodeCaseHandler {
    public void analyze();
    public ASTNode getTree();
    public boolean isAnalyzed();
    public A newInitialFlow();
    …
}
```


The method analyze executes the analysis; getTree returns the AST that is being analysed; isAnalyzed tests whether the analysis has been performed; newInitialFlow gives the initial approximation for flow facts.

Depth-First Analysis

- Traverses the tree structure of the AST by visiting each node in a depth-first order
- Suitable for developing flow-insensitive analyses
- Default behavior implemented in the class *AbstractDepthFirstAnalysis*:



Creating a Depth-First Analysis:

- Involves 2 steps:

1. Create a concrete class by extending the class *AbstractDepthFirstAnalysis*
   a) Select a type for the analysis's data
   b) Implement the method *newInitialFlow*
   c) Implement a constructor for the class

2. Override the relevant methods of *AbstractDepthFirstAnalysis*

Depth-First Analysis: NameCollector

- Associates all names that are assigned to by an assignment statement to the statement.
- Collects in one set, all names that are assigned to
- Names are stored as strings; we use *HashSetFlowSet<String>* for the analysis's flow facts.
- Implements *newInitialFlow* to return an empty *HashSetFlowSet<String>* object.

Depth-First Analysis: NameCollector --- Cont'd

1. Create a concrete class by extending the class *AbstractDepthFirstAnalysis*

```
public class NameCollector extends
    AbstractDepthFirstAnalysis
    <HashSetFlowSet<String>> {
    private int HashSetFlowSet<String> fullSet;

    public NameCollector(ASTNode tree) {
        super(tree); fullSet = newInitialFlow();
    }
    … // defines other internal methods
}
```

Depth-First Analysis: NameCollector --- Cont'd

2. Override the relevant methods of *AbstractDepthFirstAnalysis*

```
private boolean inLHS = false;

 public void caseName(Name node)  {
    if (inLHS)
        currentSet.add(node.getID());
}
```

Depth-First Analysis: NameCollector --- Cont'd

2. Override the relevant methods of *AbstractDepthFirstAnalysis*

```
public void caseAssignStmt(AssignStmt node) {
    inLHS = true;
    currentSet = newInitialFlowSet();
    analyze(node.getLHS());
    flowSets.put(node, currentSet);
    fullSet.addAll(currentSet);
    inLHS = false;
}
```

Depth-First Analysis: NameCollector --- Cont'd

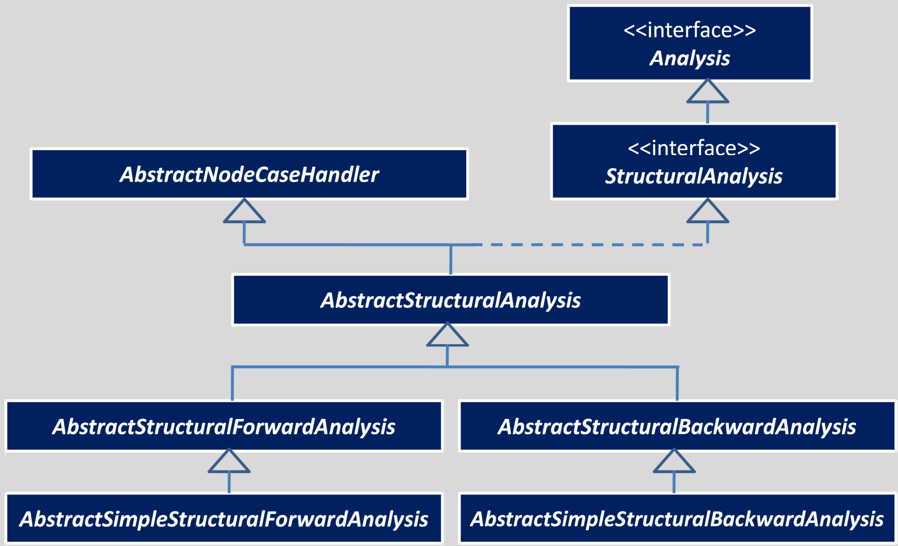
2. Override the relevant methods of *AbstractDepthFirstAnalysis*

```
public void caseParameterizedExpr
(ParameterizedExpr node) {
    analyze(node.getTarget());
}
…
```

Structural Analysis

- Suitable for developing flow-sensitive analyses
- Computes information to approximate the runtime behavior of a program.
- Provides mechanism for:
  - analyzing control structures such as *if-else*, *while* and *for* statements;
  - handling *break* and *continue* statements
- Provides default implementations for relevant methods
- May be forward or backward analysis

Structural Analysis Class Hierarchy



The interface *StructuralAnalysis*

- Extends the *Analysis* interface
- Declares more methods for structural type analysis:

```
public interface StructuralAnalysis<A extends
    FlowSet> extends Analysis<A> {
    public Map<ASTNode, A> getOutFlowSets();
    public Map<ASTNode, A> getInFlowSets();
    public void merge(A in1, A in2, A out);
    public void copy(A source, A dest);
    …
}
```

Developing a Structural Analysis

- Involves the following steps:
  1. Select a representation for the analysis's data
  2. Create a concrete class by extending the class: *AbstractSimpleStructuralForwardAnalysis* for a forward analysis and *AbstractSimpleStructuralBackwardAnalysis* for a backward analysis
  3. Implement a suitable constructor for the analysis and the method *newInitialFlow*
  4. Implement the methods *merge* and *copy*
  5. Override the relevant node case handler methods and other methods

Example: Reaching Definition Analysis

Analysis-30

Example: Reaching Definition Analysis

For every statement *s*, for every variable *v* defined by the program, compute the set of all definitions or assignment statements that assign to *v* and that *may* reach the statement *s*

A definition *d* for a variable *v* reaches a statement *s*, if there exists a path from *d* to *s* and *v* is not re-defined along that path.

Reach Def Analysis: An Implementation Step 1

Select a representation for the analysis's data:

HashMapFlowSet<String, Set<ASTNode>>

We use a map for the flow data: An entry is an ordered pair (*v*, *defs*)

where *v* denotes a variable and

defs denotes the set of definitions for *v* that may reach a given statement.

Reach Def Analysis: An Implementation Step 2

Create a concrete class by extending the class: *AbstractSimpleStructuralForwardAnalysis* for a forward analysis:

```
public class ReachingDefs extends
    AbstractSimpleStructuralForwardAnalysis
   <HashMapFlowSet<String, Set<ASTNode>>> {
   …
}
```


The analysis is a forward analysis so we extend *AbstractSimpleStructuralForwardAnalysis*

Reach Def Analysis: An Implementation Step 3

Implement a suitable constructor and the method *newInitialFlow* for the analysis:

```
public ReachingDefs(ASTNode tree) {
   super(tree);
   currentOutSet = newInitialFlow(); }

public HashMapFlowSet<String, Set<ASTNode>>
   newInitialFlow() {
   return new
   HashMapFlowSet<String,Set<ASTNode>>(); }
```

Reach Def Analysis: An Implementation Step 4a

Implement the methods *merge* and *copy*:

```
public void merge
(HashMapFlowSet<String, Set<ASTNode>> in1,
    HashMapFlowSet<String, Set<ASTNode>> in2,
    HashMapFlowSet<String, Set<ASTNode>> out) {
          union(in1, in2, out);
  }
public void
copy(HashMapFlowSet<String, Set<ASTNode>> src,
     HashMapFlowSet<String, Set<ASTNode>> dest) {
     src.copy(dest);
}
```

Reach Def Analysis: An Implementation Step 4b

```
public void
union (HashMapFlowSet<String, Set<ASTNode>> in1,
   HashMapFlowSet<String, Set<ASTNode>> in2,
   HashMapFlowSet<String, Set<ASTNode>> out) {
   Set<String> keys = new HashSet<String>();
    keys.addAll(in1.keySet()); keys.addAll(in2.keySet());
    for (String v: keys) {
       Set<ASTNode> defs = new HashSet<ASTNode>();
       if (in1.containsKey(v)) defs.addAll(in1.get(v));
       if (in2.containsKey(v)) defs.addAll(in2.get(v));
       out.add(v, defs);
    }
}
```


The helper method *union* is shown on the next slide.

Reach Def Analysis: An Implementation Step 5a

Override the relevant node case handler methods and other methods :

override caseAssignStmt(AssignStmt node)

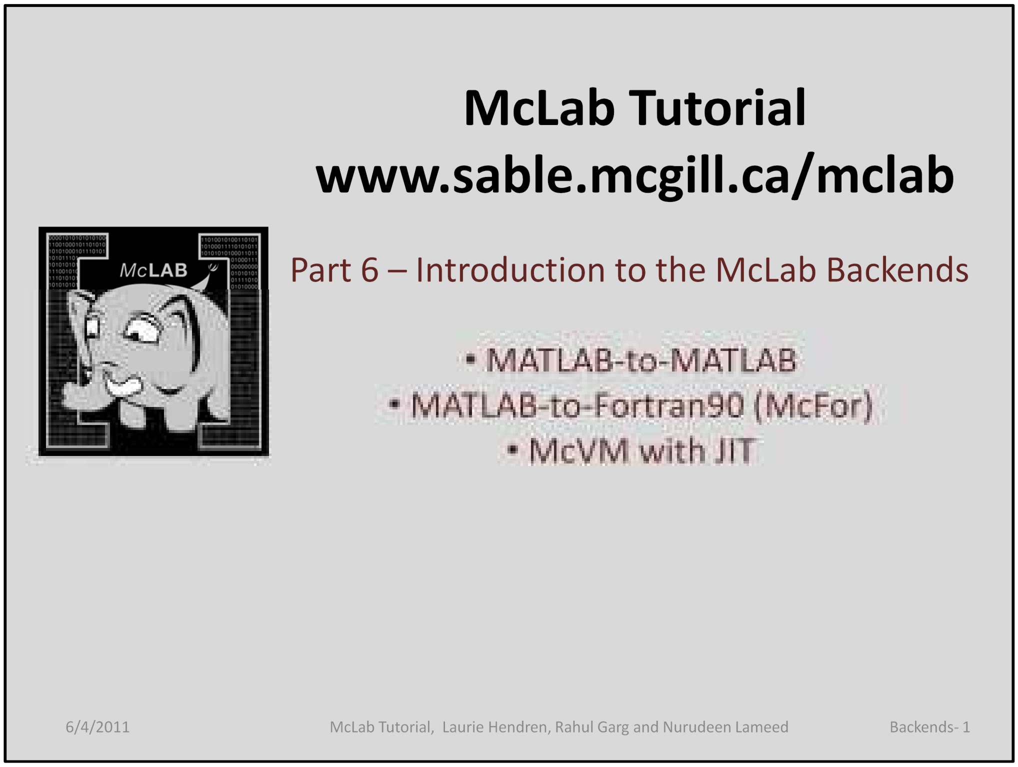
```
public void caseAssignStmt(AssignStmt node) {
    inFlowSets.put(node, currentInSet.clone() );
    currentOutSet =
        new HashMapFlowSet<String, Set<ASTNode> > ();

    copy(currentInSet, currentOutSet);
    HashMapFlowSet<String, Set<ASTNode>> gen =
        new  HashMapFlowSet<String, Set<ASTNode>> ();
    HashMapFlowSet<String, Set<ASTNode> > kill =
        new HashMapFlowSet<String, Set<ASTNode>> ();
```

Reach Def Analysis: An Implementation Step 5b

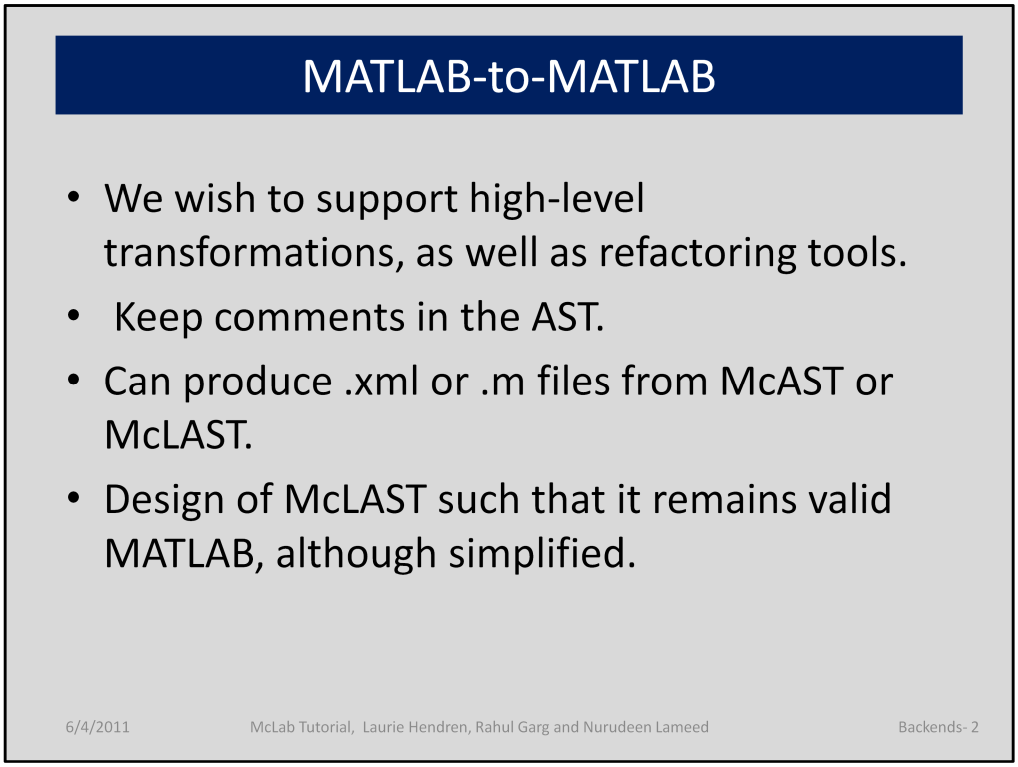
```
// compute out = (in - kill) + gen
  // compute kill
   for( String s : node.getLValues() )
    if (currentOutSet.containsKey(s))
      kill.add(s, currentOutSet.get(s));
    // compute gen
   for( String s : node.getLValues()){
     Set<ASTNode> defs = new HashSet<ASTNode>();
     defs.add(node);
     gen.add(s, defs);
   }
```

Reach Def Analysis: An Implementation Step 5c

```
    // compute (in - kill)
    Set<String> keys = kill.keySet();
    for (String s: keys)
      currentOutSet.removeByKey(s);
    // compute (in - kill) + gen
    currentOutSet = union(currentOutSet, gen);

    // associate the current out set to the node
    outFlowSets.put( node, currentOutSet.clone() );
}
```



So far in the tutorial we have concentrated on the front-end and the analysis framework. Now we turn out attention to possible backends. We support three kinds of backends, first just producing MATLAB, second a more static compiler that translates MATLAB to Fortran90, and third a Virtual machine for executing MATLAB which contains a JIT compiler.



We think that one important use of our front-end and analysis framework is for supporting high-level transformations and refactoring tools. Keeping this in mind, our AST keeps the comments from the original input program, so that we can pretty-print the transformed source, as well as the comments. McLab can generate both .xml and .m files from McAST or McLAST. We use the .xml format as a way of conveying the AST to McVM. Since we wanted to be able to generate valid MATLAB from our ASTs, we have designed the ASTs to use only valid MATLAB constructs.

MATLAB-to-Fortran90

- MATLAB programmers often want to develop their prototype in MATLAB and then develop a FORTRAN implementation based on the prototype.
- 1st version of McFOR implemented by Jun Li as M.Sc. thesis.
  - handled a smallish subset of MATLAB
  - gave excellent performance for the benchmarks handled
  - provided good insights into the problems needed to be solved, and some good initial solutions.
- 2nd version of McFOR currently under development.
  - fairly large subset of MATLAB, more complete solutions
  - provide a set of analyses, transformations and IR simplifications that will likely be suitable for both the FORTRAN generator, as well as other HLL.
- e-mail hendren@cs.mcgill.ca to be put on the list of those interested in McFor.

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Backends- 3

McVM-McJIT

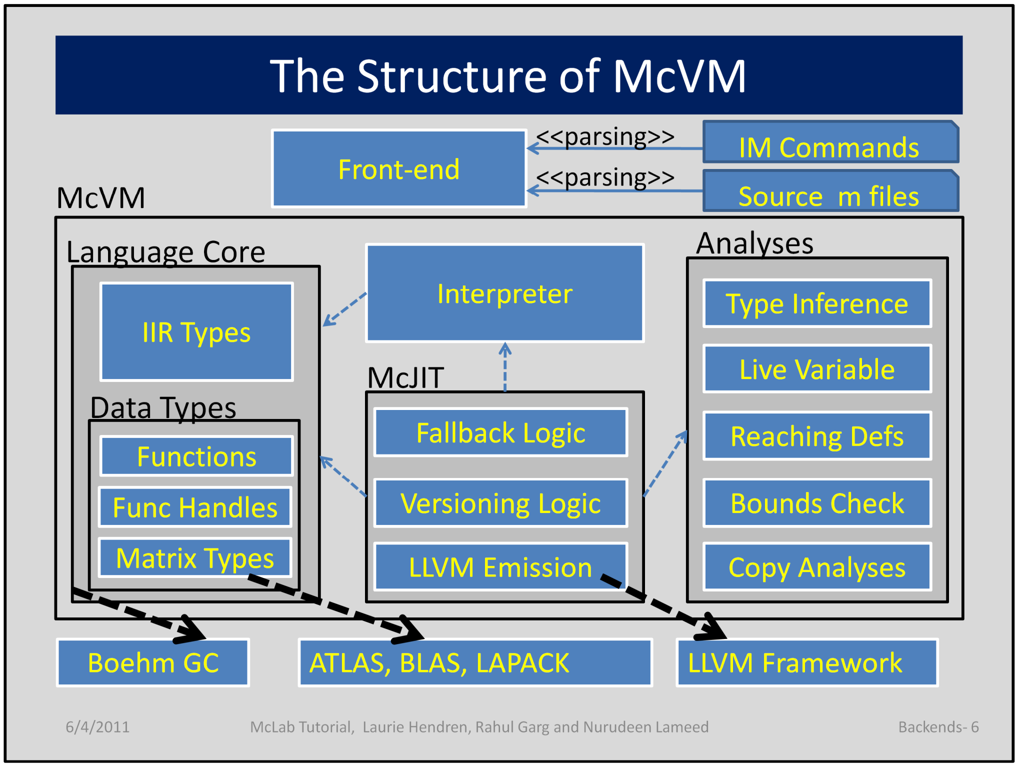
- Whereas the other back-ends are based on static analyses and ahead-of-time compilation, the dynamic nature of MATLAB makes it more suitable for a VM/JIT.
- MathWorks' implementation does have a JIT, although technical details are not known.
- McVM/McJIT is an open implementation aimed at supporting research into dynamic optimization techniques for MATLAB.

McVM is designed for flexibility and high performance.

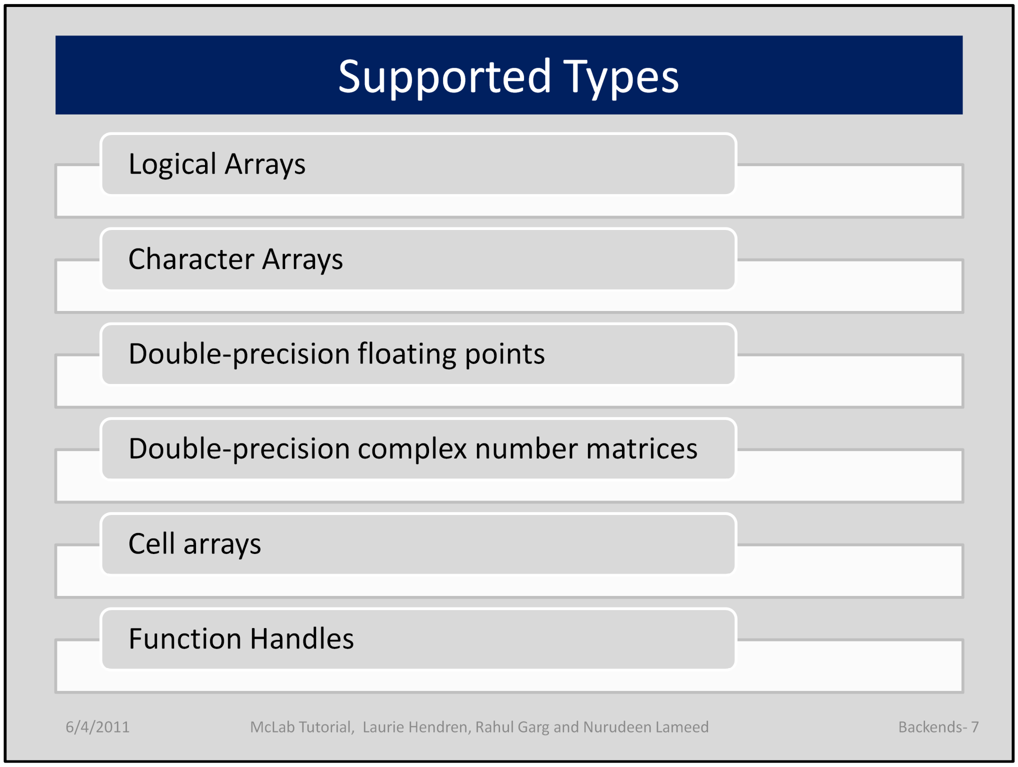
McVM Design

- A basic but fast interpreter for the MATLAB language
- A garbage-collected JIT Compiler as an extension to the interpreter
- Easy to add new data types and statements by modifying only the interpreter.
- Supported by the LLVM compiler framework and some numerical computing libraries.
- Written entirely in C++; interface with the McLab front-end via a network port.

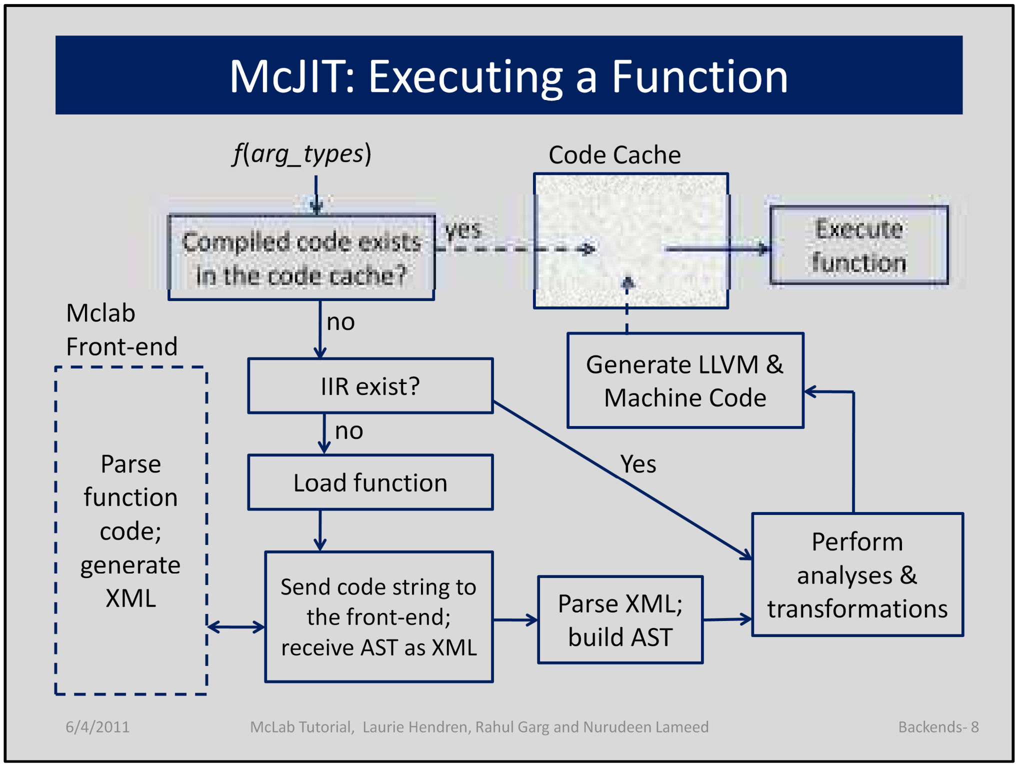
McVM is designed for flexibility and high performance.



The front-end is responsible for parsing matlab commands and m files. The language core is supported by Boehm GC, ensuring an automatic garbage collection of IIR nodes. The operations on Matrix–type data are supported by ATLAS, BLAS and LAPACK library. McJIT is supported for efficient code compilation by several analyses.



These are the currently supported types in McVM.



How does the JIT compiler execute a function? Given a function called with some argument types; McVM checks if a compiled code exists that match the call, in terms of the types of the arguments and proceed as shown.

Type Inference

- It is a key performance driver for the JIT Compiler:
  - the type information provided are used by the JIT compiler for function specialization.

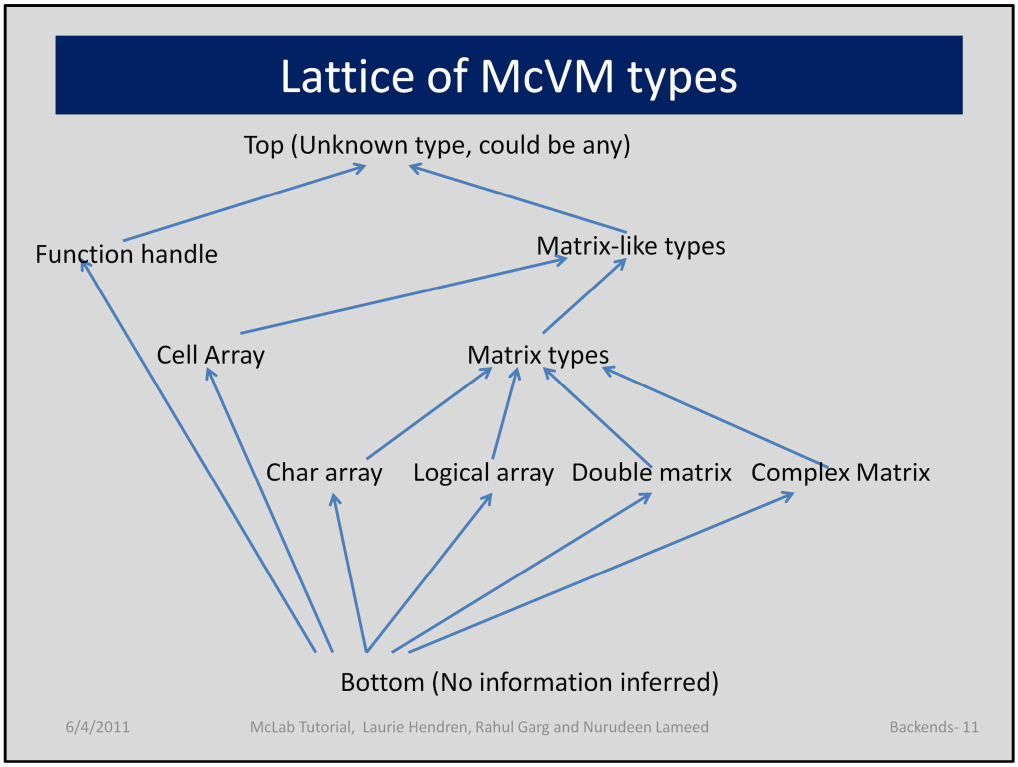
A key analysis performed McVM is the type inference analysis …

Type Inference

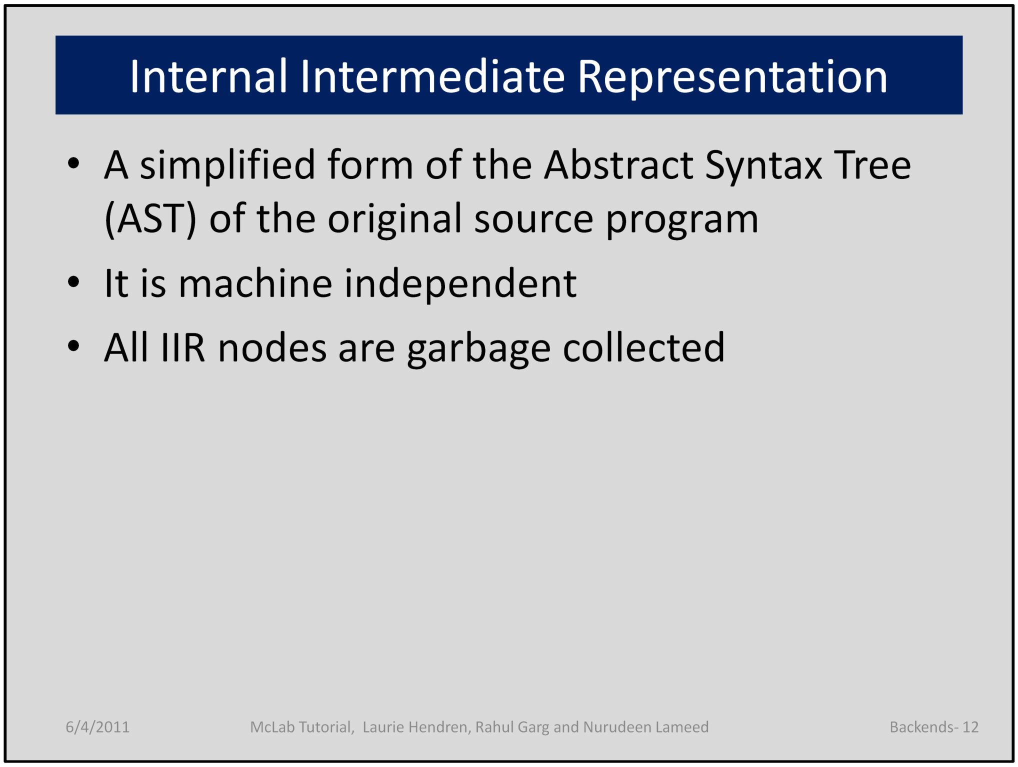
- It is a forward flow analysis: propagates the set of possible types through every possible branch of a function.
- Assumes that:

  for each input argument *arg*, there exist some possible types
- At every program point *p*, infers the set of possible types for each variable
- May generate different results for the same function at different times depending on the types of the input arguments

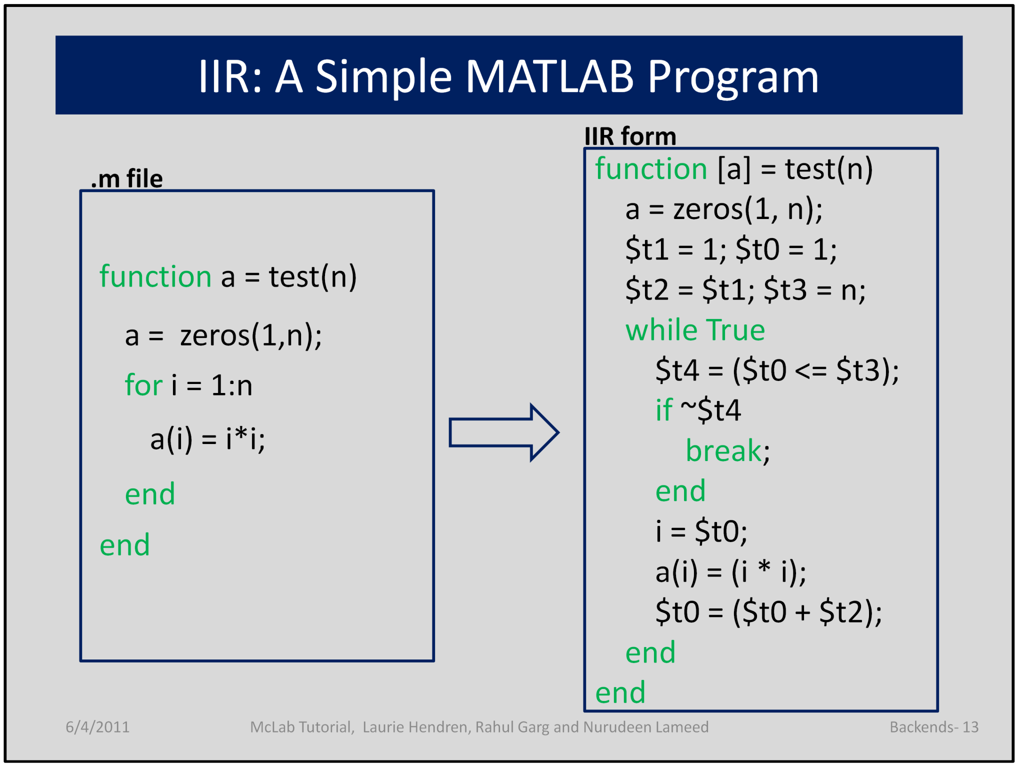
Unlike other type-inference analysis that assumes that all program components can be loaded at once and performs a whole-program analysis, our type inference analysis is intra-procedural since dynamic loading suggests that not all program components may be loaded at once. Variables can also have different types at different points in a function.



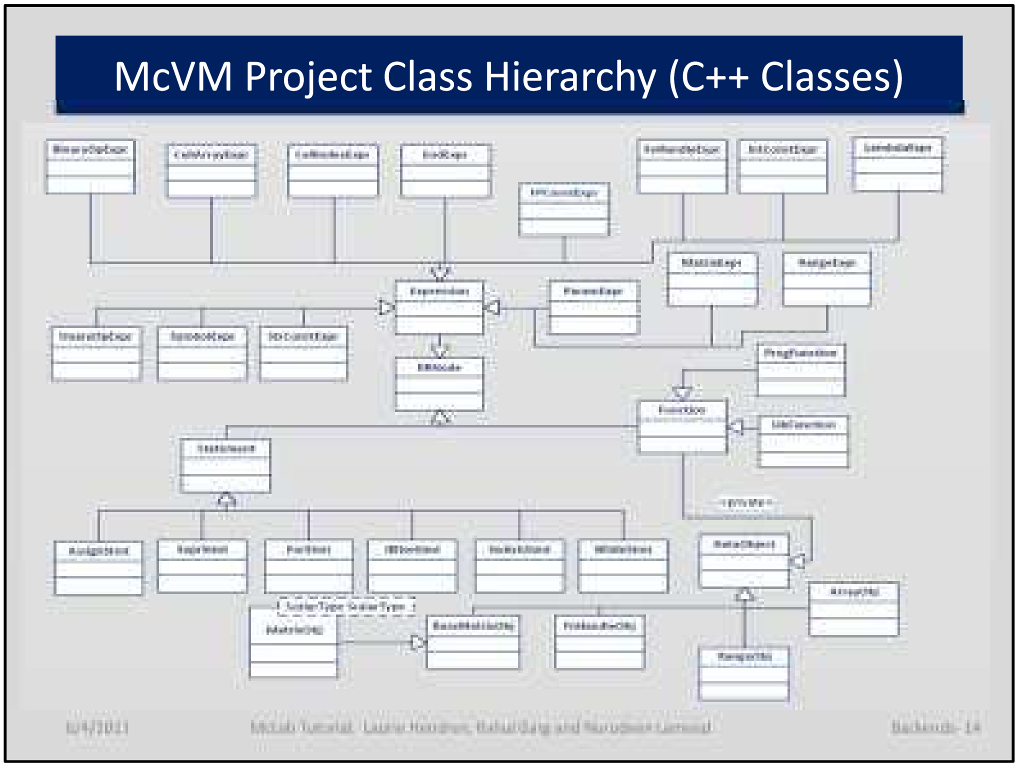
A key analysis performed McVM is the type inference analysis …



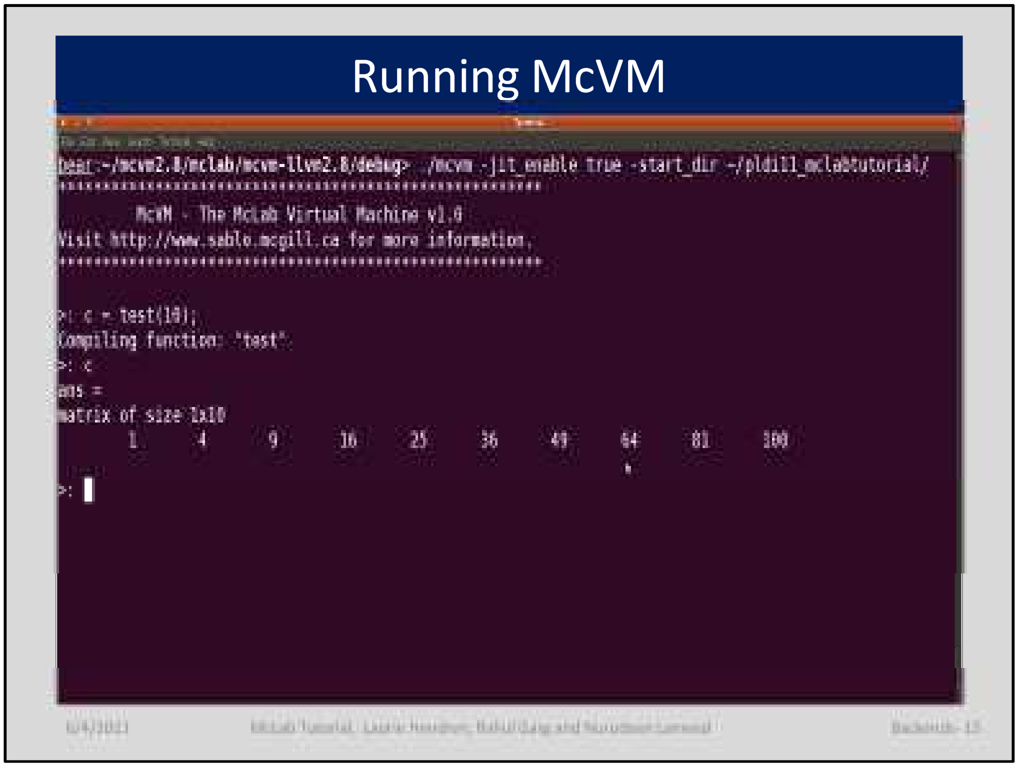
McVM converts source code into a form more amenable to analyses. It is similar in form to a three address code.



The box on the left-hand side shows the code of an .m file. This is transformed into the box on the right : the corresponding function in internal IR form.



The figure shows a simplified UML class diagram for the McVM project. At the root of class hierarchy is the IIRNode. There are a number of statements and expressions as well.



Here, I show how to start and execute functions McVM/McJIT.

McLab Tutorial
www.sable.mcgill.ca/mclab

Part 7 – McVM implementation example: if/else construct

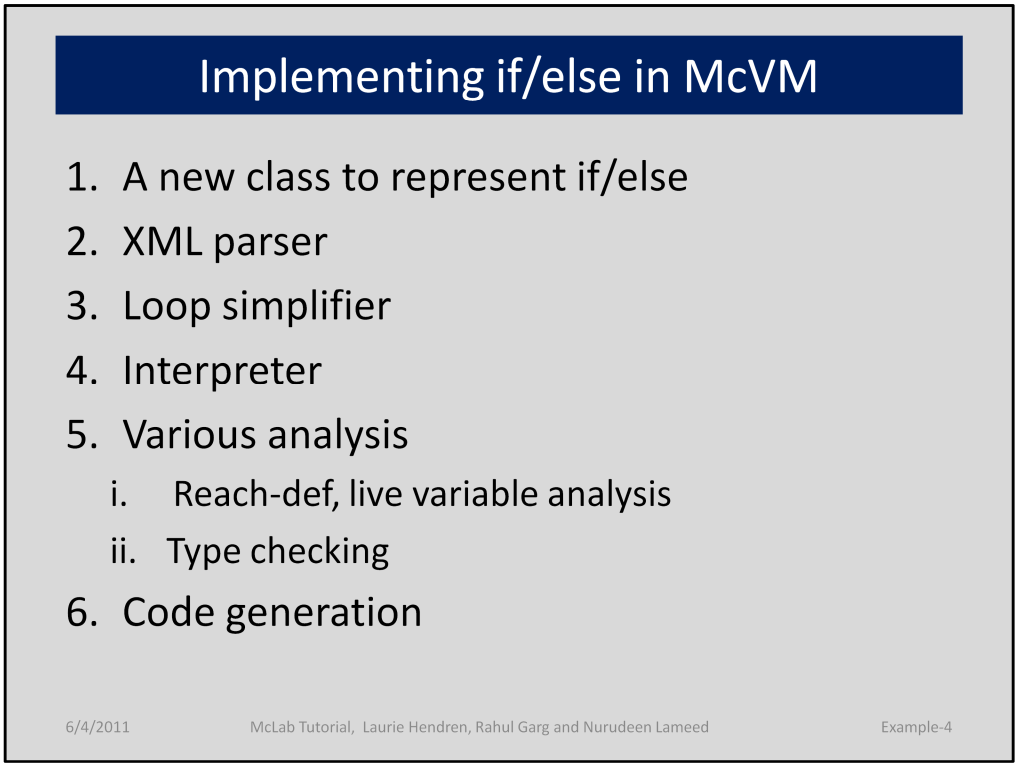
- Implementation in interpreter
- Implementation in JIT compiler

Before we start

- McVM is written in C++, but “clean” C++ ☺
- Nearly everything is a class
- Class names start in capital letters
- Typically one header and one implementation file for each class
- Method names are camel cased (getThisName)
- Members are usually private and named m_likeThis

Before we start …

- Makefile provided
  - Handwritten, very simple to read or edit
- Scons can also be used
- ATLAS/CLAPACK is not essential. Alternatives:
  - Intel MKL, AMD ACML, any CBLAS + Lapacke (eg. GotoBLAS2 + Lapacke)
- Use your favourite development tool
  - I use Eclipse CDT, switched from Vim
- Virtualbox image with everything pre-installed available on request for private use



The amount of effort required to implement each step is unequal. Step 5 is the largest most complicated step.

1. A class to represent If/Else

- Class IfElseStmt
- We will derive this class from “Statement”
- Form two files: ifelsestmt.h and ifelsestmt.cpp
- Need fields to represent:
  - Test expression
  - If body
  - Else body

Ifelsestmt.h

- class IfElseStmt: public Statement
- Methods:
  - copy(), toString(), getSymbolUses(), getSymbolDefs()
  - getCondition(), getIfBlock(), getElseBlock()
- Private members:
  - Expression *m_pCondition;
  - StmtSequence *m_pIfBlock;
  - StmtSequence *m_pElseBlock;

Modify statements.h

- Each statement has a field called m_type
- This contains a type tag
- Tag used throughout compiler for switch/case
- enum StmtType{

```
        IF_ELSE,
        SWITCH,
        FOR,
        ….
};
```

2. Modify XML Parser

- Look in parser.h, parser.cpp
- Before anything happens, must parse from XML generated by frontend
- XML parser is a simple recursive descent parser
- Add a case to parseStmt()
  - Look at the element name in the XML
  - If it is “IfStmt”, it is a If/Else
- Write a parseIfStmt() function

3. Modify transform loops

- McVM simplifies for-loops to a lower level construct
- To achieve this, we need to first find loops
- Done via a depth first search in the tree
- So add a case to this search to say:
  - Search in the if block
  - Search in the else block
  - Return
- transform_loops.cpp

4. Add to interpreter

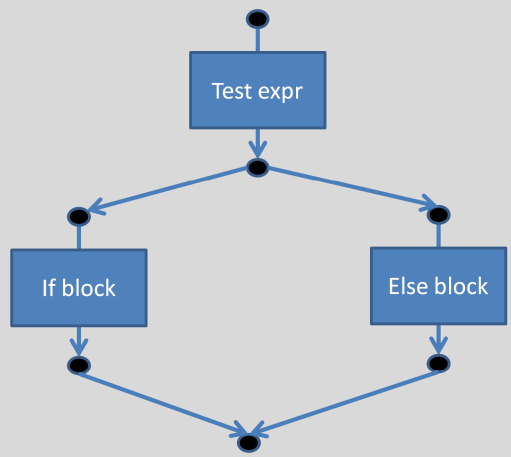
- Always implement in interpreter before implementing in JIT compiler
- It is a simple evaluator: no byte-code tricks, no direct-threaded dispatch etc.
- Add a case to statement evaluation:
  - Evaluate test condition
  - If true, evaluate if block
  - If false, evaluate else block
- interpreter.cpp :
  - Case in execStatement()
  - Calls evalIfElseStmt()

Moment of silence .. Or review

- At this point, if/else has been implemented in the interpreter
- If you don't enable JIT compilation, then you can now run if/else
- Good checkpoint for testing and development

Flow analysis recap

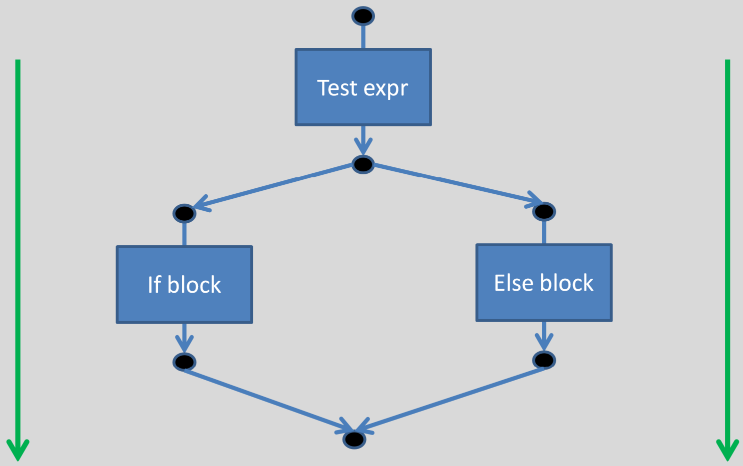
- Compute program property at each program point



Flow analysis recap

- We want to compute property at each program point
- Typically want to compute a map of some kind at each program point
- Program points are not inside statements, but just before and after
- Usually unions computed at join points
- Can be forward or backwards depending on the analysis

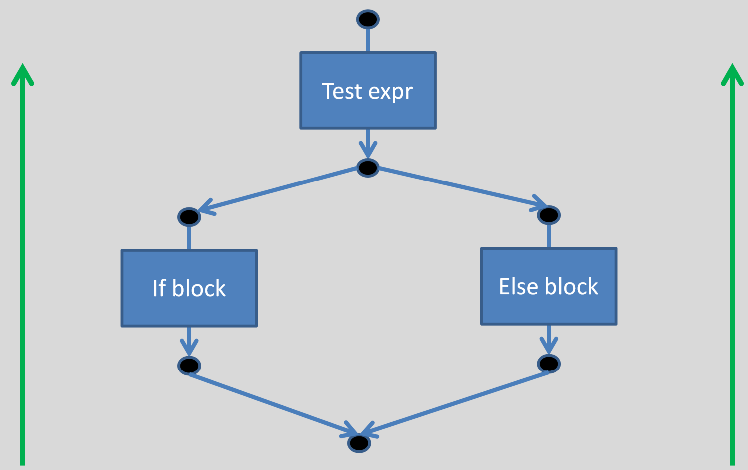
Reaching definitions analysis



McVM reach-defs analysis

- Look in analysis_reachdefs (.h/.cpp)
- getReachDefs() is an overloaded function to compute reach-defs
- ReachDefInfo class to store analysis info
- If/Else:
  - Record reach-defs for test expression
  - Compute reach-defs for if and else blocks by calling getReachDefs() for StmtSequence
  - Compute union at post-if/else point

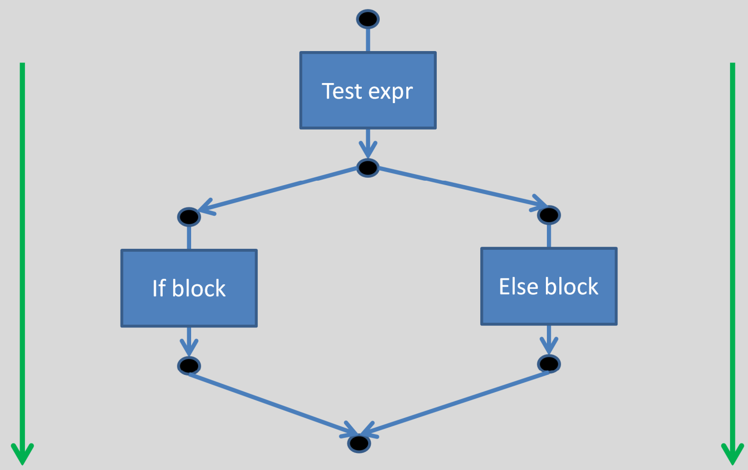
Live variable analysis



McVM live vars analysis

- Look in analysis_livevars (.h/.cpp)
- getLiveVars() is an overloaded function
- LiveVarInfo is a class to store live-vars info
- If/Else:
  - Information flows backwards from post-if/else
  - Flow live-vars through the if and else blocks
  - Compute union at post-test expression
  - Record live-vars info of test expression

Type inference analysis



Type inference

- Look in analysis_typeinfer (.h/.cpp)
- inferTypes() is an overloaded function to perform type inference for most node-types
- For If/else:
  - Infer type of test expression
  - Infer type of if and else blocks
  - Merge information at post-if/else point

Flow analysis tips

- We define a few typedefs for data structures like maps, sets
  - eg: VarDefSet: typedef of set of IIRNode* with appropriate comparison operators and allocator
- When trying to understand flow analysis code, start from code for assignment statements
- Pay attention to statements like return and break

Code generation and LLVM

- LLVM is based upon a typed SSA representation
- LLVM can either be accessed through a C++ API, or you can generate LLVM byte-code directly
- We use the C++ API
- Much of the complexity of the code generator due to SSA representation required by LLVM
- However, we don't do an explicit SSA conversion pass

Code generation in McVM

- SSA conversion is not explicitly represented in the IR
- SSA conversion done while doing code generation
- Assignment instructions are usually not generated directly if Lvalue is a symbol
- In SSA form, values of expressions are important, not what they are assigned to
- We store mapping of symbols to values in an execution environment

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed Example-22

A complete explaination is probably out of the scope of the tutorial. You are referred to compAssignStmt() in jitcompiler.cpp for details.

Compiling if/else

- Four steps:
  - Compile test expression
  - Compile if block (compStmtSeq)
  - Compile else block (compStmtSeq)
  - Call matchBranchPoints() to do appropriate SSA book-keeping at merge point
- Rest of the code is book-keeping for LLVM
- Such as forming proper basic blocks when required

McLab Tutorial
www.sable.mcgill.ca/mclab

Part 8 – Wrap Up

- Summary
- Ongoing and Future Work
- Further Sources

Thanks for attending the tutorial.

Tutorial Summary

- MATLAB is a popular language and an important PLDI research area.
- McLab aims to provide tools to support such research.
  - Front-end: extensible scanner, parser, attributes
    - example extension: AspectMatlab
  - IR and analysis framework:
    - two levels of IR, high-level McAST and lower-level McLAST
    - structure-based flow analysis framework
  - Back-ends: MATLAB, McVM with McJIT and McFor

Ongoing and Future Work

- MATLAB refactoring tools:
  - code cleanup
  - refactoring towards Fortran generation
  - include static call graph and interprocedural analysis framework
- MATLAB extensions:
  - AspectMatlab
  - Typing Aspects

Back-end (McVM/McJIT)

- On-stack replacement
- Dynamic optimizations – correct choice of inlining and basic block positioning.
- Optimizations for multicore systems
- Compilation to GPUs and mixed CPU/GPU systems
- Portability and performance across multiple CPU and GPU families

Where to look for more info

- www.sable.mcgill.ca
  - /software
    - currently have McVM and AspectMatlab on the web site
    - can ask for McLab front-end and analysis framework, we will also add to the web site soon
  - /publications
    - papers and thesis, in particular
    - MetaLexer (Andrew Casey)
    - McLab Front-end and Analysis Framework (Jesse Doherty)
    - McVM (Maxime Chevalier-Boisvert)
    - McFor (1^{st} version Jun Li, 2^{nd} version Anton Dubrau)
    - tutorials, starting with this one

6/4/2011 McLab Tutorial, Laurie Hendren, Rahul Garg and Nurudeen Lameed, Part 8 Wrap Up - 5

Keep in Touch

- main web site:
http://www.sable.mcgill.ca/mclab
- mailing list:
mclab-list@sable.mcgill.ca
- bug reports:
https://svn.sable.mcgill.ca/mclab-bugzilla/
- people:
hendren@cs.mcgill.ca, rahul.garg@mail.mcgill.ca, nurudeen.lameed@mail.mcgill.ca

Contact us!